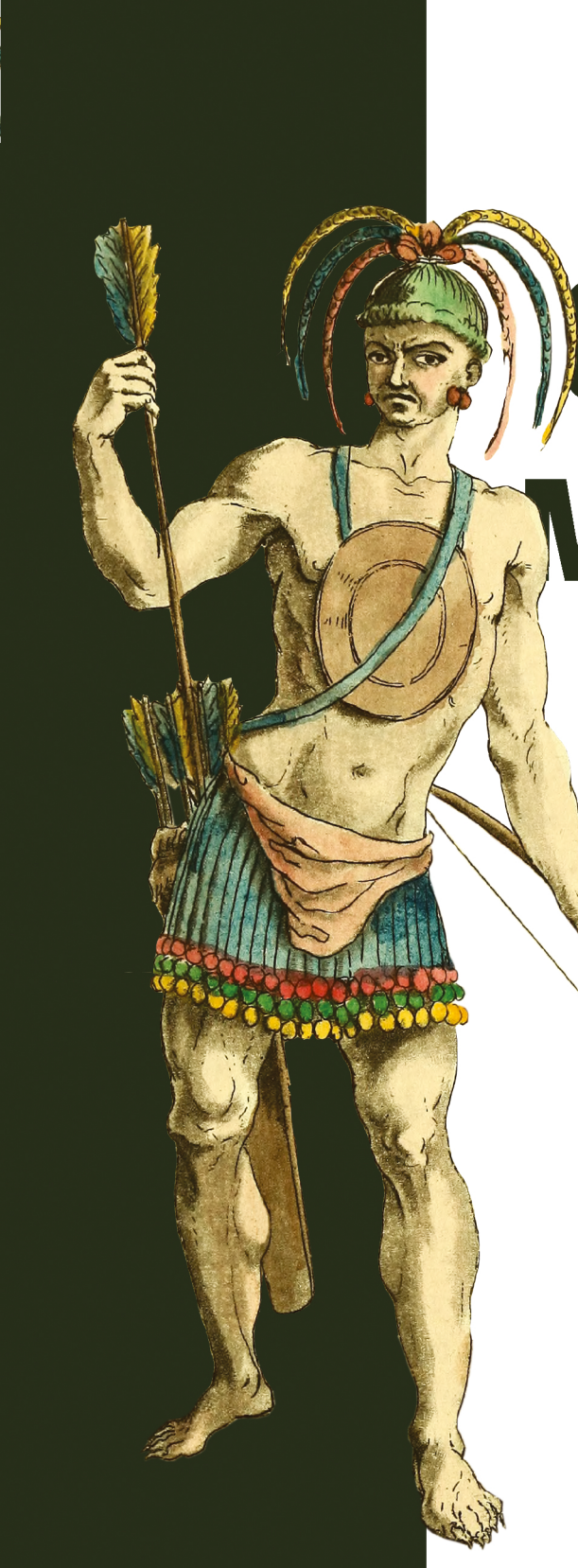
адаптировано для JAVA 11



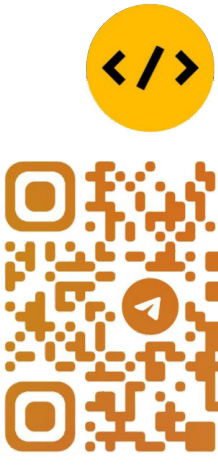
Система модулей Java

Николай Парлог
предисловие Кевлина Хенни

MANNING

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

The Java Module System



NICOLAI PARLOG

Foreword by Kevlin Henney

MANNING

SHELTER ISLAND

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Николай Парлог
предисловие Кевлина Хенни

Система модулей Java

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2021

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ББК 32.973.2-018.1
УДК 004.438
П18

Парлог Николай

П18 Система модулей Java. — СПб.: Питер, 2021. — 464 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1620-1

Создать надежное и безопасное приложение гораздо проще, если упаковать код в аккуратные блоки. Система модулей в Java представляет собой языковой стандарт для создания таких блоков. Теперь вы можете контролировать взаимодействия различных JAR и легко обнаруживать недостающие зависимости. Фундаментальные изменения архитектуры затронули ядро Java, начиная с версии 9. Все API ядра распространяются в виде модулей, а для библиотек, фреймворков и приложений аналогичный подход можно считать хорошей практикой и рекомендацией.

Вы освоите наилучшие практики модульного проектирования, отладки приложения и его развертывания перед сдачей в продакшен.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.438

Права на издание получены по соглашению с Apress. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617294280 англ.
ISBN 978-5-4461-1620-1

© 2019 by Manning Publications Co. All rights reserved
© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Для профессионалов», 2021
© Павлов А., перевод с английского языка, 2020

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Краткое содержание

Предисловие ... 16
Вступление ... 18
Благодарности ... 20
О книге ... 22
Об авторе ... 31
Об обложке ... 32

Часть I. Привет, модули

Глава 1. Первый элемент головоломки ... 35
Глава 2. Структура модульного приложения ... 67
Глава 3. Определение модулей и их свойств ... 87
Глава 4. Построение модулей от исходного кода до JAR ... 124
Глава 5. Запуск и отладка модульных приложений ... 143

Часть II. Адаптация под реальные проекты

Глава 6. Проблемы совместимости при переходе на Java 9 и выше ... 167
Глава 7. Повторяющиеся проблемы при переходе на Java 9 и выше ... 186
Глава 8. Постепенная модуляризация существующих проектов ... 213
Глава 9. Стратегии миграции и модуляризации ... 243

Часть III. Расширенные функции системы модулей

Глава 10. Использование сервисов для разделения модулей ... 271
Глава 11. Уточнение API и зависимостей ... 300
Глава 12. Рефлексия в модульном мире ... 329
Глава 13. Версии и модули: возможное и невозможное ... 362
Глава 14. Настройка образа среды выполнения с помощью jlink ... 378
Глава 15. Собираем все вместе ... 406

Приложения

Приложение А. Резюмируем путь класса ... 436
Приложение Б. Обзор API рефлексии ... 439
Приложение В. Наблюдение за JVM с унифицированным ведением журнала ... 443
Приложение Г. Анализ зависимостей проекта с помощью JDeps ... 450
Приложение Д. Ориентация на несколько версий Java с мультиверсионными JAR-файлами ... 458

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Оглавление

Предисловие ... 16
Вступление ... 18
Благодарности ... 20
О книге ... 22
Кому стоит прочитать эту книгу ... 22
Структура книги ... 23
Части и главы ... 23
Выберите собственный путь ... 24
Предостережение ... 26
О коде ... 26
О версии Java ... 27
Оформление листингов ... 28
Оформление имен методов ... 28
Заместители во фрагментах кода ... 29
Команды и их вывод ... 29
Дискуссионный форум liveBook ... 29
От издательства ... 30
Об авторе ... 31
Об обложке ... 32

Часть I. Привет, модули

Глава 1. Первый элемент головоломки ... 35
1.1. Что такое модульность ... 36
1.1.1. Визуализация ПО в виде графов ... 37
1.1.2. Влияние принципов проектирования ... 39
1.1.3. Что такое модульность ... 40
1.2. Уничтожение модулей до Java 9 ... 41
1.3. Трудности до Java 9 ... 44
1.3.1. Невыраженные зависимости между JAR-файлами ... 45

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

1.3.2. Затенение классов с одинаковыми именами ... 46
1.3.3. Конфликты разных версий на одном проекте ... 48
1.3.4. Сложная загрузка класса ... 49
1.3.5. Слабая инкапсуляция в JAR-файлах ... 49
1.3.6. Проверки безопасности должны быть выполнены вручную ... 51
1.3.7. Низкая производительность при загрузке ... 51
1.3.8. Негибкая среда выполнения в Java ... 52
1.4. Взгляд на модули с высоты птичьего полета ... 52
1.4.1. Все на свете — это модули ... 52
1.4.2. Ваш первый модуль ... 55
1.4.3. Система модулей в действии ... 55
1.4.4. Ваш немодульный проект в целом будет в порядке ... 60
1.5. Цели системы модулей ... 61
1.5.1. Надежная настройка: не оставлять после себя JAR ... 62
1.5.2. Надежная инкапсуляция: делаем внутренний код модуля недоступным ... 62
1.5.3. Автоматическая безопасность и улучшенное удобство сопровождения ... 63
1.5.4. Улучшенная производительность при запуске ... 63
1.5.5. Масштабируемая платформа Java ... 63
1.5.6. Не цели ... 64
1.6. Навыки, старые и новые ... 64
1.6.1. Чему вы научитесь ... 65
1.6.2. Что нужно знать ... 65
Резюме ... 66

Глава 2. Структура модульного приложения ... 67
2.1. Знакомство с ServiceMonitor ... 68
2.2. Модуляризация ServiceMonitor ... 72
2.3. Разделение ServiceMonitor на модули ... 72
2.4. Организация файлов в структуре каталогов ... 73
2.5. Объявление и описание модулей ... 74
2.5.1. Объявление зависимостей от других модулей ... 76
2.5.2. Объявление публичных API модуля ... 76
2.5.3. Визуализация ServiceMonitor с помощью модульного графа ... 77
2.6. Компиляция и упаковка модулей ... 79
2.7. Запуск ServiceMonitor ... 81
2.8. Расширение модульной кодовой базы ... 81
2.9. Разбор полетов: эффекты системы модулей ... 82
2.9.1. Что система модулей делает для вас ... 82
2.9.2. Что еще модульная система может сделать для вас ... 84
2.9.3. Добавление необязательных зависимостей ... 86
Резюме ... 86

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Глава 3. Определение модулей и их свойств ... 87
3.1. Модули: строительные блоки модульных приложений ... 88
3.1.1. Модули Java (JMOD), поставляемые с JDK ... 88
3.1.2. Модульные файлы JAR: доморощенные модули ... 89
3.1.3. Декларация модулей: определение свойств модулей ... 90
3.1.4. Множество типов модулей ... 95
3.2. Читабельность: соединяем части вместе ... 97
3.2.1. Достижение надежной настройки ... 98
3.2.2. Эксперименты с ненадежными настройками ... 99
3.3. Доступность: объявление публичных API ... 105
3.3.1. Обеспечение надежной инкапсуляции ... 107
3.3.2. Инкапсуляция переходных зависимостей ... 109
3.3.3. Столкновения инкапсуляций ... 110
3.4. Путь модуля: позвольте Java узнать о модулях ... 114
3.4.1. Регулирование модулей: анализ и верификация структуры приложения ... 115
3.4.2. Граф модуля: представление архитектуры приложения ... 117
3.4.3. Добавление модулей в граф ... 120
3.4.4. Добавление ребер в граф ... 121
3.4.5. Доступность — постоянная цель ... 122
Резюме ... 122

Глава 4. Построение модулей от исходного кода до JAR ... 124
4.1. Структурирование проекта с помощью каталогов ... 125
4.1.1. Новое предложение — новое соглашение? ... 125
4.1.2. Созданная структура каталогов ... 126
4.1.3. Место для деклараций модулей ... 127
4.2. Компиляция отдельного модуля ... 128
4.2.1. Компиляция модульного кода ... 128
4.2.2. Модульное или немодульное? ... 129
4.3. Компиляция нескольких модулей ... 131
4.3.1. Безыскусный подход ... 132
4.3.2. Путь к исходнику модуля: информирование компилятора о структуре проекта ... 132
4.3.3. Звездочка как токен для имени модуля ... 133
4.3.4. Несколько путей к исходникам модулей ... 135
4.3.5. Настройка начального модуля ... 136
4.3.6. Стоит ли оно того? ... 137
4.4. Параметры компилятора ... 138
4.5. Упаковка модульного JAR-файла ... 139
4.5.1. Быстрый обзор jar ... 139
4.5.2. Анализ JAR ... 140

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

4.5.3. Объявление точки входа ... 141
4.5.4. Параметры архиватора ... 141
Резюме ... 142

Глава 5. Запуск и отладка модульных приложений ... 143
5.1. Запуск JVM с модулями ... 144
5.1.1. Определение основного класса ... 144
5.1.2. Что, если начальный модуль и основной модуль — не один и тот же ... 145
5.1.3. Передача параметров приложению ... 146
5.2. Загрузка ресурсов из модулей ... 147
5.2.1. Загрузка ресурсов до Java 9 ... 148
5.2.2. Загрузка ресурсов, начиная с Java 9 и позже ... 148
5.2.3. Загрузка ресурсов за пределами модуля ... 150
5.3. Отладка модулей и модульных приложений ... 151
5.3.1. Анализ отдельных модулей ... 152
5.3.2. Проверка набора модулей ... 152
5.3.3. Проверка модульного графа ... 153
5.3.4. Перечисление обозреваемых модулей и зависимостей ... 154
5.3.5. Исключение модулей во время разрешения ... 156
5.3.6. Наблюдение за системой модулей с помощью ведения журнала ... 159
5.4. Параметры виртуальной машины Java ... 162
Резюме ... 164

Часть II. Адаптация под реальные проекты

Глава 6. Проблемы совместимости при переходе на Java 9 и выше ... 167
6.1. Работа с модулями JEE ... 169
6.1.1. Что особенного в модулях JEE? ... 170
6.1.2. Разрешение модулей JEE вручную ... 171
6.1.3. Вмешиваемся в сторонние реализации модулей JEE ... 172
6.2. Приведение к URLClassLoader ... 173
6.2.1. Загрузчики классов приложения, тогда и сейчас ... 173
6.2.2. Обходимся без URLClassLoader ... 175
6.2.3. Поиск проблемных приведений ... 176
6.3. Обновленный макет каталога образов среды выполнения ... 176
6.4. Выбор, замена и расширение платформы ... 179
6.4.1. Компактных профилей больше нет ... 179
6.4.2. Удален механизм расширения ... 180
6.4.3. Удален механизм переопределения утвержденных стандартов ... 180
6.4.4. Удалены некоторые параметры пути к загрузочному классу ... 180

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

6.4.5. Компиляции для Java 5 больше нет ... 180
6.4.6. Удален выбор версии JRE ... 181
6.5. Мелочи, которые приводят к большим неприятностям ... 181
6.5.1. Новый формат версий ... 182
6.5.2. Исчезновение инструментов ... 183
6.5.3. Всякие мелочи ... 183
6.5.4. Новые устаревшие элементы в Java 9, 10 и 11 ... 184
Резюме ... 184

Глава 7. Повторяющиеся проблемы при переходе на Java 9 и выше ... 186
7.1. Инкапсуляция внутренних API ... 187
7.1.1. Внутренние API под микроскопом ... 188
7.1.2. Анализ зависимостей с помощью JDeps ... 192
7.1.3. Компиляция с помощью внутренних API ... 194
7.1.4. Выполнение с помощью внутренних API ... 195
7.1.5. Параметры компилятора и JVM для доступа к внутренним API ... 200
7.2. Улучшение разделенных пакетов ... 202
7.2.1. Что не так с разделенными пакетами ... 203
7.2.2. Эффекты разделенных пакетов ... 204
7.2.3. Множество способов справиться с разделенными пакетами ... 208
7.2.4. Исправление модулей: последнее средство для обработки разделенных пакетов ... 209
7.2.5. Поиск разделенных пакетов с помощью JDeps ... 210
7.2.6. Примечание о конфликте версий зависимостей ... 211
Резюме ... 211

Глава 8. Постепенная модуляризация существующих проектов ... 213
8.1. Почему наращивание модульности необязательно ... 214
8.1.1. Если бы требовалось, чтобы каждый JAR был модульным ... 215
8.1.2. Смешивание и соединение простых JAR-файлов с модулями ... 215
8.1.3. Технические основы наращиваемой модульности ... 217
8.2. Безымянный модуль, он же путь к классу ... 218
8.2.1. Хаос пути к классам, захваченного безымянным модулем ... 220
8.2.2. Разрешение модулей для безымянного модуля ... 221
8.2.3. Зависимость от безымянного модуля ... 223
8.3. Автоматические модули: простые JAR в пути модуля ... 225
8.3.1. Имена автоматических модулей: мелочь, имеющая большое значение ... 227
8.3.2. Разрешение модулей для автоматических модулей ... 230
8.3.3. Все в автоматических модулях? ... 238
8.3.4. Зависимость от автоматических модулей ... 239
Резюме ... 242

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Глава 9. Стратегии миграции и модуляризации ... 243
9.1. Стратегии миграции ... 244
9.1.1. Подготовительные обновления ... 244
9.1.2. Оценка усилий ... 245
9.1.3. Непрерывная интеграция в Java 9+ ... 246
9.1.4. Мысли о параметрах командной строки ... 250
9.2. Стратегии модуляризации ... 253
9.2.1. Восходящая модуляризация: когда все зависимости — модульные ... 255
9.2.2. Нисходящая модуляризация: когда нет времени дожидаться всех зависимостей ... 256
9.2.3. Модуляризация изнутри наружу: если проект находится в середине стека ... 257
9.2.4. Применение данных стратегий к проекту ... 258
9.3. Делаем JAR-файлы модульными ... 259
9.3.1. Открытые модули как промежуточный шаг ... 259
9.3.2. Генерация модульных деклараций с помощью JDeps ... 260
9.3.3. Взлом сторонних JAR-файлов ... 263
9.3.4. Публикация модульных JAR-файлов для Java 8 и выше ... 265
Резюме ... 267

Часть III. Расширенные функции системы модулей

Глава 10. Использование сервисов для разделения модулей ... 271
10.1. Изучение потребности в сервисах ... 272
10.2. Сервисы в модульной системе платформы Java ... 274
10.2.1. Использование, предоставление и потребление услуг ... 274
10.2.2. Разрешение модулей для сервисов ... 279
10.3. Эффективное проектирование сервисов ... 282
10.3.1. Типы, которые могут стать сервисами ... 283
10.3.2. Использование фабрик в качестве сервисов ... 283
10.3.3. Изоляция потребителей от глобального состояния ... 285
10.3.4. Организация сервисов, потребителей и поставщиков в модули ... 288
10.3.5. Использование сервисов для разбиения циклических зависимостей ... 289
10.3.6. Объявление сервисов в разных версиях Java ... 291
10.4. Доступ к сервисам с помощью ServiceLoader API ... 294
10.4.1. Загрузка и доступ к сервисам ... 294
10.4.2. Особенности загрузки сервисов ... 296
Резюме ... 297

Глава 11. Уточнение API и зависимостей ... 300
11.1. Подразумеваемая читабельность: передача зависимостей ... 301
11.1.1. Выявление зависимостей модуля ... 302

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

11.1.2. Модификатор transitive: подразумеваемая читабельность зависимости ... 304
11.1.3. Когда использовать подразумеваемую читабельность ... 306
11.1.4. Когда стоит полагаться на подразумеваемую читабельность ... 307
11.1.5. Рефакторинг модулей с подразумеваемой читабельностью ... 309
11.1.6. Рефакторинг модулей путем их слияния ... 313
11.2. Необязательные зависимости ... 314
11.2.1. Загадка надежной конфигурации ... 315
11.2.2. Модификатор static: маркировка зависимостей как необязательных ... 316
11.2.3. Разрешение модулей для необязательных зависимостей ... 318
11.2.4. Написание кода с необязательной зависимостью ... 319
11.3. Квалифицированный экспорт: ограничение доступа к конкретным модулям ... 321
11.3.1. Предоставление внутренних API ... 322
11.3.2. Экспорт пакетов в модули ... 323
11.3.3. Когда использовать квалифицированный экспорт ... 325
11.3.4. Экспорт пакетов в командной строке ... 327
Резюме ... 327

Глава 12. Рефлексия в модульном мире ... 329
12.1. Почему директивы exports не подходят для рефлексии ... 331
12.1.1. Врываемся в немодульный код ... 332
12.1.2. Принудительная публикация внутренних типов ... 332
12.1.3. Квалифицированный экспорт создает связь с определенными модулями ... 333
12.1.4. Нет поддержки глубокой рефлексии ... 333
12.2. Открытые пакеты и модули: предназначены для использования с рефлексией ... 334
12.2.1. Открытие пакетов для доступа в режиме выполнения ... 335
12.2.2. Открытие пакетов для определенных модулей ... 336
12.2.3. Экспорт и открытие пакетов ... 337
12.2.4. Открытие модулей: закрытие рефлексии ... 338
12.3. Рефлексия модулей ... 339
12.3.1. Обновление рефлективного кода для модулей (или нет) ... 340
12.3.2. Использование обработчика переменных вместо рефлексии ... 342
12.3.3. Анализ свойств модуля с помощью рефлексии ... 344
12.3.4. Изменение свойств модуля с помощью рефлексии ... 347
12.3.5. Пересылка открытых пакетов ... 348
12.4. Динамическое создание диаграмм модулей со слоями ... 349
12.4.1. Что такое слои ... 350
12.4.2. Анализ слоев ... 352
12.4.3. Создание модульных слоев ... 355
Резюме ... 359

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Глава 13. Версии и модули: возможное и невозможное ... 362
13.1. Отсутствие поддержки версий в JPMS ... 363
13.1.1. Нет поддержки нескольких версий ... 363
13.1.2. Нет поддержки выбора версии ... 366
13.1.3. Что может принести будущее ... 368
13.2. Запись информации о версии ... 369
13.2.1. Запись версий при сборке модулей ... 369
13.2.2. Доступ к версиям модуля ... 370
13.3. Запуск нескольких версий модуля в отдельных слоях ... 372
13.3.1. Зачем нужен стартер для раскрутки дополнительных слоев ... 373
13.3.2. Раскручивание слоев для вашего приложения, Apache Twill и Cassandra Java Driver ... 374
Резюме ... 377

Глава 14. Настройка образа среды выполнения с помощью jlink ... 378
14.1. Создание пользовательских образов среды выполнения ... 379
14.1.1. Начало работы с jlink ... 380
14.1.2. Содержание и структура образа ... 381
14.1.3. Включение сервисов в образы среды выполнения ... 382
14.1.4. Правильно подобранные образы с помощью jlink и jdeps ... 385
14.2. Создание автономных образов приложений ... 387
14.2.1. Включение модулей приложений в образы ... 388
14.2.2. Создание собственного средства запуска приложения ... 391
14.2.3. Безопасность, производительность и стабильность ... 393
14.3. Генерация образов в операционных системах ... 393
14.4. Использование плагинов jlink для оптимизации образов ... 395
14.4.1. Плагины для jlink ... 395
14.4.2. Уменьшение размера образа ... 398
14.4.3. Улучшение производительности во время выполнения ... 402
14.5. Параметры для jlink ... 403
Резюме ... 404

Глава 15. Собираем все вместе ... 406
15.1. Добавляем навороты в ServiceMonitor ... 406
15.1.1. Разнообразные зависимости ... 410
15.1.2. Снижение видимости ... 411
15.1.3. Отделение от сервисов ... 411
15.1.4. Загрузка кода со слоями во время выполнения ... 412
15.1.5. Обработка зависимостей от простых JAR-файлов ... 412
15.2. Советы для разработчиков модульных приложений ... 413
15.2.1. Модуль или нет? ... 413
15.2.2. Идеальный модуль ... 414

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

15.2.3. Позаботьтесь о декларациях модуля ... 419
15.2.4. Взлом кода путем редактирования модульных деклараций ... 422
15.3. Технологический ландшафт ... 424
15.3.1. Maven, Gradle и другие инструменты сборки ... 425
15.3.2. OSGi ... 427
15.3.3. Микросервисы ... 431
15.4. Размышления о модульной экосистеме ... 433
Резюме ... 434

Приложения

Приложение А. Резюмируем путь класса ... 436
А.1. Использование пути к классам для загрузки JAR-файлов приложений ... 436
А.2. Путь к классам, начиная с Java 9 ... 437

Приложение Б. Обзор API рефлексии ... 439
Б.1. Основные типы и методы ... 441
Б.2. Взлом API с помощью setAccessible ... 442
Б.3. Аннотации помечают код для рефлексии ... 442

Приложение В. Наблюдение за JVM с унифицированным ведением журнала ... 443
В.1. Что такое унифицированное ведение журнала ... 444
В.2. Определяем, какие сообщения должны отображаться ... 444
В.3. Определяем, куда вывести сообщения ... 447
В.4. Определяем, что выводить в сообщениях ... 447
В.5. Настройка всего конвейера регистрации ... 448

Приложение Г. Анализ зависимостей проекта с помощью JDeps ... 450
Г.1. Знакомство с JDeps ... 451
Г.2. Добавление зависимостей в анализ ... 452
Г.3. Настройка вывода JDeps ... 453
Г.4. Детализация в зависимости от вашего проекта ... 454
Г.5. JDeps понимает модули ... 456

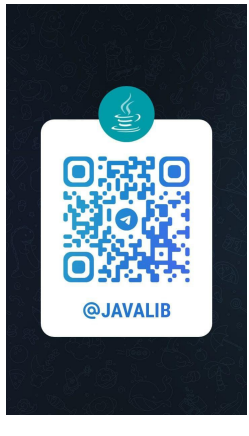
Приложение Д. Ориентация на несколько версий Java с мультиверсионными JAR-файлами ... 458
Д.1. Создание мультиверсионного JAR ... 459
Д.2. Внутренняя работа MB-JAR ... 461
Д.3. Рекомендации по использованию ... 461
Д.3.1. Организация исходного кода ... 461
Д.3.2. Организация байт-кода ... 462
Д.3.3. Когда использовать MB-JAR ... 462

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Разве это не прекрасно, что все происходит по его желанию?[1]

Посвящается Габи и Майе

Ещё больше книг по Java в нашем телеграм канале:
https://t.me/javalib



[1] Перевод отрывка из песни No Leaf Clover группы Metallica.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Предисловие

Стремление к модульности не ново. В 1968 году на знаковой конференции НАТО, посвященной программной инженерии, сыгравшей ключевую роль в популяризации видения программных компонентов и самого термина *«разработка ПО»* (software engineering), Э. Э. Дэвид (E. E. David) изложил подход к разработке больших систем:

> Определите подмножества системы, достаточно малые, чтобы ими можно было управлять, затем постройте из них подсистему. Эта стратегия подразумевает создание системы на основе модулей, которые могут быть написаны, протестированы и изменены без учета их взаимодействия друг с другом.

На этой же конференции Х. Р. Джиллетт (H. R. Gillette) рассказал, как модульность стала предпосылкой эволюции систем:

> Модульность помогает изолировать функциональные элементы системы. Один модуль можно отладить, улучшить или расширить с минимальным взаимодействием или разрывом системы.

Ничего нового. Просто понадобилось немного практики и несколько языков программирования, прежде чем эта тема была поднята и изучена.

История модульности Java, подобно фрагментам головоломки, разбросана во времени и пространстве и закодирована. Первой и существенной реализацией модулей в Java стал *класс*. В других языках он являл собой смесь модульности и типов данных, предоставляя типам данных, их свойствам и методам некоторую приватность и связность. Язык Java сделал следующий шаг, превратив класс из артефакта в самостоятельный двоичный компонент.

Увы, если попытаться ответить на вопрос: «Что означает *быть достаточно малым, чтобы им можно было управлять?*» — класс оказывается не просто, а чересчур малым. Как заметили Маркс и Энгельс в 1848 году:

> История всех доселе существующих обществ — это история классовой борьбы.
>
> *Манифест Коммунистической партии*

За исключением совсем небольших кодовых баз и очень загроможденных классов, класс — не самая большая часть компонентной архитектуры.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

История Java также связана с пустыми обещаниями насчет *пакета* — во всех смыслах данного слова — чего-то, что запечатано и отправлено, целостно, неделимо и готово к использованию. Вот только это переросло в кое-что не имеющее ничего общего с первоначальным замыслом: пространства имен для организации кода в папках — в открытый, но нескромный и обреченный на провал план, запятнанный и отягощенный модными в свое время, но в итоге непрактичными доменными именами.

Затем наступил момент Пандоры — появление мифа. Греческий миф о Пандоре, девушке, приносящей дары, обычно искажают: якобы она открывает ящик, содержимое которого наводит болезни на все человечество. Так вот, это был не ящик, а сосуд. То, что открыла Пандора, — *pithos* (сосуд), впоследствии неверно переведенный как *pyxis* (ящик). Как и при написании кода, названия имеют значение.

Появление JAR-файлов стало первым шагом в сторону компонентной модели, но за ним не последовало ничего больше чем архивирование файлов классов. Для борьбы с возникающим JAR-адом многие средства (возможно, самые наглядные — это инструменты сборки и OSGi-контейнеры) расширили JAR-модель и ее манифест, что позволило продвинуться дальше на пути к модульности.

Но все это в прошлом. А что же сейчас? Что ждет нас в будущем?

Ответ прямо перед вами. Именно поэтому вы и читаете данную книгу.

Java 9 объединил множество фрагментов головоломки в один с помощью модулей — системы, вплетенной прямо в ядро платформы, а не в расширение за ее пределами. Система модулей Java не имеет никаких отсылок к прошлому. Она обязана не только сохранить богатство уже существующего кода, чтобы не превращать в хаос действующие экосистемы, но и одновременно предложить нечто для проектов, которым еще только предстоит быть написанными в этом постоянно меняющемся мире.

Природу модулей и зависимостей, детали синтаксиса и компонентизации необходимо понять на механическом уровне. С точки зрения проектирования стоит знать все за и против работы с модулями. Как и в случае с любой другой концепцией, модульность — это не волшебный соус, который можно просто добавить в разработку; она требует осторожности, навыков и внимания. Вам нужно знать ответы на вопросы о том, что произойдет с уже существующим кодом в модульном мире; о том, как это повлияет на развертывание и разработку; и даже о том, о чем вы еще не задумывались.

Вопросов очень много. Вот почему вы сейчас читаете данную книгу.

Николай способен ответить на эти и другие вопросы. Он начал отслеживать модули с того момента, как они появились на горизонте. Он нырнул прямо в глубину и преодолел мелководье JSR и реализаций. Он углубился в такие детали, куда вам, может быть, и не придется заглянуть. Его тщательность и внимание к мелочам позволят вам овладеть всей полнотой знаний — от теории к практике, от уровня новичка до гуру.

Это книга-дар. Открывайте, читайте, наслаждайтесь.

Кевлин Хенни (Kevlin Henney), Curbralan

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Вступление

Я познакомился с системой модулей ранним утром в апреле 2015 года. Перед работой, проверяя входящие сообщения в OpenJFX, я наткнулся на сообщение от пользователя JavaFX, который был обеспокоен тем, что приватные API стали недоступными «из-за ограничений модульности». Я помню, как подумал: «Не может быть. Java ни за что не пошел бы на такое несовместимое изменение». Списав это на недоразумение, я отправился на работу.

Затем, после обеда, мы немного поспорили с коллегой. Ничего особенного, но из-за появившегося раздражения я решил вернуться домой пораньше и насладиться солнечным весенним днем. Выйдя на балкон со стаканом прохладного пива, я решил почитать что-нибудь. Но что именно? Из любопытства я начал просматривать ответы на сообщение, которое я пролистал сегодня утром, — и это меня очень увлекло!

Всю следующую неделю я буквально пожирал любой обрывок информации, какой только смог найти, о *Project Jigsaw* — базе, на основе которой развивалась система модулей. Оказалось, что опасения пользователя JavaFX вовсе не были беспочвенными.

Сначала я сосредоточился только на том, что могло выйти из строя в Java 9. Намечающаяся выгода пока еще казалась несколько туманной. К счастью, в то время я работал над крупным Java-проектом и постепенно начал понимать, как можно использовать систему модулей для улучшения и сопровождения его общей структуры. Все больше и больше частей складывалось в единую картину, и спустя пару недель я загорелся идеей ввести модули в экосистему — даже если бы это что-нибудь нарушило.

Переход от опасений по поводу совместимости к пониманию ценности, которую модули могут внести в проект, — самый распространенный вариант. Но это еще не все! Помимо беспокойства об уже существующих кодовых базах, вы можете захотеть ввести систему модулей в только-только стартовавший проект либо заинтересоваться изучением растущего влияния модулей на экосистему. Куда бы вы ни направились, данная книга станет вашим путеводителем.

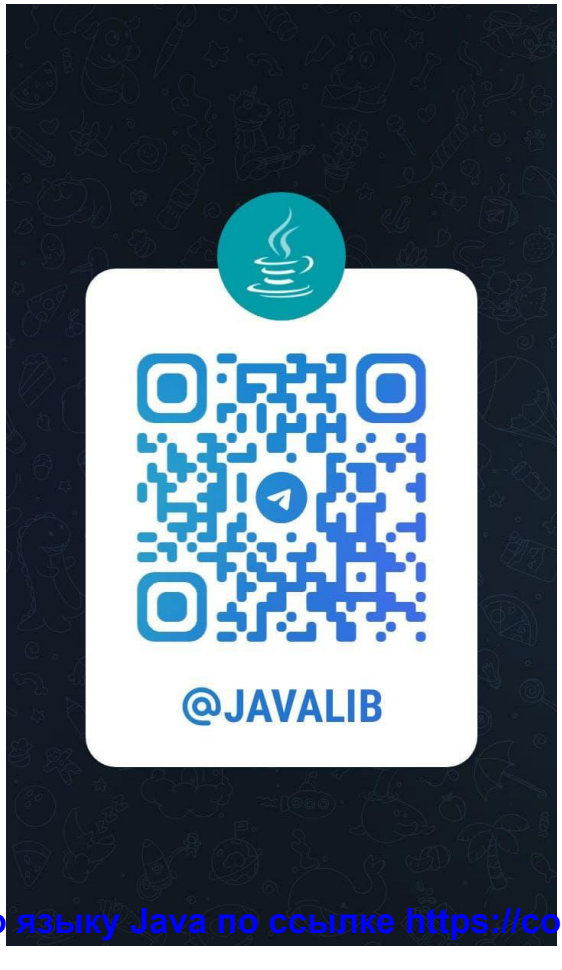
Если вам интересно, куда приведет это путешествие, то вспомните Java 8. Введенные в нем лямбда-выражения стали чем-то гораздо более важным, чем просто новая особенность языка, — они глубоко повлияли на сообщество и экосистему. Они позна-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

комили миллионы Java-разработчиков с основами функционального программирования и стали началом путешествия, которое открыло нам глаза на новые концепции и сделало нас более сильными программистами. Они также повлияли на создание новых библиотек и даже кое-чему научили уже существующие фреймворки.

Имейте это в виду, размышляя о системе модулей. Это больше, чем просто термин, — он приглашает вас в путешествие, в котором вы узнаете о модульности во всех ее проявлениях, научитесь правильно проектировать и сопровождать большие программные проекты, а также более активно использовать модульность в библиотеках, фреймворках и инструментах.

Ещё больше книг по Java в нашем телеграм канале:
https://t.me/javalib



Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Благодарности

Прежде всего я хочу поблагодарить Марину Майклс (Marina Michaels), моего главного редактора в Manning. Без ее доброты, настойчивости, профессионализма и чувства юмора эта книга не вышла бы никогда. Марина постоянно направляла меня на моем писательском пути. Что еще более важно, она снова и снова помогала мне сконцентрироваться и написать пару новых глав.

По этой же причине я хочу отказаться от благодарностей в адрес создателей *Civilization* и *Stellaris*, *«Во все тяжкие»* и *«Экспансии»*, авторов многих выдающихся научно-фантастических книг, прочитанных мной за последние годы, а также любого человека на американском ночном телевидении: было бы здорово наслаждаться плодами вашего труда, но мне пришлось потратить бо́льшую часть этого времени на то, чтобы трудиться самому.

Хотелось бы выделить еще трех человек из Manning: Джеанну Боярски (Jeanne Boyarsky), которая наладила со мной важную техническую обратную связь и занималась данной книгой вплоть до самого ее выпуска, когда я уже не мог этого делать; Вишеслава Радовича (Viseslav Radovic), проявившего бесконечное терпение при создании иллюстраций и подгонке их под мой вкус; а также Жан-Франсуа Морина (Jean-François Morin), который безустанно просматривал книгу и примеры кода в поисках ошибок. Вот еще несколько человек, вовлеченных в превращение почти миллиона писем, разбросанного по куче Asciidoc-файлов, в настоящую книгу: Шерил Вейсман (Cheryl Weisman), Рэйчел Герберт (Rachael Herbert), Дэвид Новак (David Novak), Тиффани Тейлор (Tiffany Taylor), Мелоди Долаб (Melody Dolab), Happenstance Type-O-Rama, Александар Драгосавлевич (Aleksandar Dragosavljević), Мэри Пирджис (Mary Piergies) и Мария Тюдор (Marija Tudor). Наконец, я хочу поблагодарить всех рецензентов, которые нашли время, чтобы прочитать мою книгу и оставить к ней комментарии: Анто Аравинта (Anto Aravinth), Бориса Василе (Boris Vasile), Кристиана Кройцер-Бека (Christian Kreutzer-Beck), Конора Редмонда (Conor Redmond), Гаурава Тули (Gaurav Tuli), Джанкарло Массари (Giancarlo Massari), Гвидо Пио Мариотти (Guido Pio Mariotti), Ивана Милосавлевича (Ivan Milosavljević), Джеймса Райта (James Wright), Джереми Брайана (Jeremy Bryan), Кэтлин Эстрада (Kathleen Estrada), Марию Джемини (Maria Gemini), Марка Дечампса (Mark Dechamps), Миккеля Арентофта (Mikkel Arentoft), Рамбабу Поса (Rambabu Posa), Себастьяна Чеха (Sebastian Czech), Шобху Айера (Shobha Iyer), Стива Доусонн-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Андоха (Steve Dawsonn-Andoh), Татьяну Фесенко (Tatiana Fesenko), Томаша Борека (Tomasz Borek) и Тони Свитса (Tony Sweets). Спасибо всем вам!

Кроме того, я благодарен всем людям из сообщества и Oracle, которые приняли участие в разработке системы модулей. Не только за то, что без их усердного труда просто не было бы ничего, о чем стоило бы писать, но и за качественную документацию и интересные разговоры. Я хочу отдельно упомянуть Марка Рейнхолда (Mark Reinhold), Алекса Бакли (Alex Buckley) и Алана Бейтмана (Alan Bateman) за распространение наших совместных усилий, а также за ответы на многочисленные вопросы, которыми я засыпал почтовый ящик Project Jigsaw. Спасибо вам!

Среди тех, кто, возможно, не ждет благодарности, я хотел бы поблагодарить Роберта Крюгера (Robert Krüger), который, сам того не ведая, вызвал во мне интерес к системе модулей тем самым роковым письмом от 8 апреля 2015 года; Кристиана Глёклера (Christian Glökler) за спор, впервые приведший меня к Project Jigsaw в тот день; и Бориса Терзиса (Boris Terzic) за то, что всегда вдохновлял меня на следующий шаг (и за разрешение поиграться с каждой новой версией Java на работе). Затем — всех тех прекрасных людей, которые обратились ко мне за последний год и оставили отзывы обо всем, что я уже написал, — ваш всепоглощающий позитив подарил мне кучу энергии. Спасибо вам!

Всем моим друзьям: спасибо, что были рядом, несмотря на то что у меня было так мало времени, и спасибо за вдохновение и поддержку на протяжении всего пути. Моей семье: я очень старался, чтобы работа над книгой не отнимала то драгоценное время, которое мы проводим вместе, — спасибо, что прощали меня, когда мне это не удавалось. Без вашего бесконечного терпения, любви и поддержки всего этого не было бы. Я бы не стал тем, кто я есть, без вас. Я люблю вас.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

О книге

Java 9 ввел систему модулей для платформы Java в язык и экосистему, сделав примитивы модульности легкодоступными всем Java-разработчикам. Для многих, включая меня, такая концепция нова, поэтому в данной книге обучение начинается с нуля. Мы пройдем путь от самых основ к постижению расширенных функций языка. Более того, книга поможет обновить ваши существующие проекты до Java 9+, постепенно наращивая их модульность.

Обратите внимание: мы не собираемся лишь изучать модульность как таковую. Это сложная тема, на которую написаны отдельные книги (к примеру, *Java Application Architecture* Кирка Кнорншильда (Kirk Knoernschild, Prentice Hall, 2012). Однако в процессе введения модульности в действие вы просто не сможете избежать изучения причин, по которым это вообще стоит делать.

Кому стоит прочитать эту книгу

Система модулей — интересный зверь. Насколько просты ее основные принципы и концепции — настолько же сложно ее влияние на экосистему. Она не удивит сию же секунду, подобно лямбда-выражениям, но ее воздействие на экосистему невозможно недооценить. Однако сейчас все это вряд ли имеет значение. В настоящее время модульная система — такая же часть Java, как компилятор, модификатор `private` и оператор `if`, и каждый разработчик обязан знать и понимать ее так же, как все перечисленное.

К счастью, начинать всегда просто. В основу системы модулей заложено всего несколько простых концепций, которые поймет любой разработчик, хоть немного знающий Java. В целом вы поймете эту систему, если уже знаете принципы работы модификаторов доступа, имеете представление, как использовать `javac`, `jar` и `java`, и знаете, что JVM загружает классы из JAR-файлов.

Если все описанное — о вас и вы любите новые испытания, то я приглашаю прочитать эту книгу. Возможно, вы не сразу расставите все точки над i, но по крайней мере, приобретете четкое понимание системы модулей и многих других фактов, которые сложатся в ваше собственное видение экосистемы Java.

С другой стороны, чтобы расставить все точки над i, нужно иметь многолетний опыт участия в разработке Java-проектов. Если коротко, то чем обширнее эти про-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

екты и чем больше вы вовлечены в развитие их архитектуры, выбор верных зависимостей и борьбу с ошибками, тем больше вы оцените вклад системы модулей в них. И тогда же проще будет определить влияние модульности на ваш проект и экосистему в целом.

Структура книги

Данная книга разбита на несколько уровней. Прежде всего, она разделена на главы (и три большие части), но вам не обязательно изучать их от начала до конца. У меня есть пара предложений, что читать и в каком порядке.

Части и главы

Книга состоит из 15 глав, разделенных на три части.

Часть I, *«Привет, модули»*, показывает недостатки Java, для устранения которых и была придумана система модулей, а также объясняет ее базовые механизмы вкупе с принципами создания, построения и запуска модульных приложений.

- Глава 1, *«Первый элемент головоломки»*, рассказывает о недостатках поддержки Java модульности на уровне JAR, их негативных последствиях и о том, как система модулей намерена устранить эти недостатки.
- Глава 2, *«Структура модульного приложения»*, показывает, как спроектировать и запустить модульное приложение, и приводит примеры программы, которая будет использоваться на протяжении всей книги. Данная глава обрисует цельную картину, но не будет вдаваться в детали — за нее это сделают остальные главы.
- Глава 3, *«Определение модулей и их свойств»*, описывает модули и основные блоки, из которых и строятся модули, а также то, как система модулей обрабатывает их для достижения приоритетной цели — построения надежного и удобного в сопровождении проекта.
- Глава 4, *«Построение модулей от исходного кода до JAR»*, показывает, как скомпилировать и собрать модульный проект с помощью команд `javac` и `jar`.
- Глава 5, *«Запуск и отладка модульных приложений»*, исследует множество новых опций команды `java`. Запустить модульное приложение достаточно легко, поэтому большая часть данной главы посвящена инструментам, которые понадобятся вам для поиска и устранения неполадок.

Часть II, *«Адаптация под реальные проекты»*, отклоняется от использования полностью модульных приложений и предназначена для перевода существующих проектов на Java 9+ и их постепенной модуляризации.

- Глава 6, *«Проблемы совместимости при переходе на Java 9 и выше»*, исследует наиболее распространенные препятствия, с которыми вы можете столкнуться при переводе действующей кодовой базы на Java 9 (и это еще до создания модулей).
- Глава 7, *«Повторяющиеся проблемы при переходе на Java 9 и выше»*, обсуждает еще два препятствия, вынесенные отдельно, — они не ограничены миграцией,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

и потому вы можете столкнуться с ними даже после того, как обновите и модуляризуете проект.

- Глава 8, *«Постепенная модуляризация существующих проектов»*, показывает, как взять большой проект, запущенный на Java 9, и перевести его в модули. Хорошая новость — вам не придется делать это за раз.
- Глава 9, *«Стратегии миграции и модуляризации»*, основывается на предыдущих главах и рассматривает стратегии, которые помогут вам перенести и модуляризовать существующую кодовую базу.

Часть III, *«Расширенные функции системы модулей»*, демонстрирует возможности, основанные на базисе, полученном в части I.

- Глава 10, *«Использование сервисов для разделения модулей»*, показывает, как система модулей поддерживает разделение потребителей и разработчиков API.
- Глава 11, *«Уточнение API и зависимостей»*, расширяет основные механизмы зависимости и доступности, рассмотренные в главе 3, предоставляя гибкость, необходимую для разработки под хаотичные реальные сценарии.
- Глава 12, *«Рефлексия в модульном мире»*, обсуждает, как рефлексия потеряла суперсилу; какими должны быть приложения, библиотеки и фреймворки для того, чтобы она заработала; а также новые эффективные возможности, которые были добавлены в рефлексию.
- Глава 13, *«Версии и модули: возможное и невозможное»*, объясняет причины, по которым система модулей в большинстве своем игнорирует информацию о версиях, минимальную поддержку версионности в модулях, а также то, как сложно (но можно) запустить разные версии одного и того же модуля.
- Глава 14, *«Настройка образа среды выполнения с помощью jlink»*, показывает, как можно извлечь выгоду из модуляризованного JDK, если создать собственный образ среды выполнения, включающий только необходимые модули, а также демонстрирует плюсы модуляризованного приложения, включенного в этот образ, — самостоятельной единицы развертывания.
- Глава 15, *«Собираем все вместе»*, показывает, как выглядит приложение, представленное в главе 2, со всеми наворотами из части III. Кроме того, включает советы по оптимальному использованию системы модулей.

Выберите собственный путь

Пусть данная книга станет для вас не просто одноразовым инструментом, который научит пользоваться системой модулей, когда вы прочтете ее от корки до корки. Не то чтобы в этом было нечто плохое, но мне хочется чего-то большего. Я хочу, чтобы она стала вашим путеводителем и вы изучили по ней максимально интересные вам темы. И пусть затем она останется у вас на столе, готовая в любое время служить в качестве справочника.

Я предлагаю вам прочитать книгу от корки до корки, однако вы можете этого не делать. Я уже упомянул, что для каждого механизма и инструмента отведена отдельная глава и все подробности собраны в одном месте.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Чтобы упростить погружение в главу, я часто повторяю и сопоставляю факты, представленные в других частях книги, чтобы вы могли прочитать о них, даже пропустив соответствующий материал. Если я слишком часто повторяюсь или переборщил со ссылками, — надеюсь, вы меня простите.

На случай, если вы не книжный червь, ниже я добавил несколько путей на выбор.

У меня всего два часа — покажите, что у вас есть:

- *«Цели системы модулей»*, раздел 1.5;
- *«Структура модульного приложения»*, глава 2;
- *«Определение модулей и их свойств»*, глава 3;
- *«Советы для разработчиков модульных приложений»*, раздел 15.2.

Хочу, чтобы мое приложение работало на Java 9:

- *«Первый элемент головоломки»*, глава 1;
- *«Определение модулей и их свойств»*, глава 3;
- *«Проблемы совместимости при переходе на Java 9 и выше»*, глава 6;
- *«Повторяющиеся проблемы при переходе на Java 9 и выше»*, глава 7;
- *«Безымянный модуль, он же путь к классу»*, раздел 8.2;
- *«Стратегии миграции»*, раздел 9.1.

Я думаю о запуске нового проекта с модулями:

- *«Привет, модули»*, часть I;
- *«Использование сервисов для разделения модулей»*, глава 10;
- *«Уточнение API и зависимостей»*, глава 11;
- *«Собираем все вместе»*, глава 15.

Как система модулей изменила экосистему Java:

- *«Первый элемент головоломки»*, глава 1;
- *«Структура модульного приложения»*, глава 2;
- *«Определение модулей и их свойств»*, глава 3;
- *«Проблемы совместимости при переходе на Java 9 и выше»*, глава 6;
- *«Повторяющиеся проблемы при переходе на Java 9 и выше»*, глава 7;
- *«Расширенные функции системы модулей»*, часть III (возможно, кроме глав 10 и 11).

Меня пригласили на вечеринку. Хочу узнать пару фишек системы модулей для поддержания разговора:

- *«Взгляд на модули с высоты птичьего полета»*, раздел 1.4;
- *«Цели системы модулей»*, раздел 1.5;
- *«Структурирование проекта с помощью каталогов»*, раздел 4.1;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- *«Загрузка ресурсов из модулей»*, раздел 5.2;
- *«Отладка модулей и модульных приложений»*, раздел 5.3;
- что-нибудь из *«Проблем совместимости при переходе на Java 9 и выше»*, глава 6; и *«Повторяющихся проблем при переходе на Java 9 и выше»*, глава 7, поможет начать разговор;
- *«Версии и модули: возможное и невозможное»*, глава 13;
- *«Настройка образа среды выполнения с помощью jlink»*, глава 14.

Это круто. Хочу знать все!

- Прочитайте все подряд. Можете оставить часть II, *«Адаптация под реальные проекты»*, на потом, если у вас еще нет своего проекта.

Какой бы путь вы ни выбрали, обращайтесь к указателям, в частности, в начале и конце главы, чтобы решить, куда пойти дальше.

Предостережение

Эта книга полна новых терминов, примеров, советов и сведений, которые нужно запомнить. Чтобы упростить для вас поиск необходимого материала, я специально выделил два вида информации.

Определения *новой концепции, термина, свойства модуля* или *параметра командной строки* выделены курсивом. Особенно значимые даны в рамке с заголовком. Эти абзацы в книге — самые важные, так что к ним стоит обращаться, чтобы уяснить, как работает тот или иной механизм.

ВАЖНАЯ ИНФОРМАЦИЯ

Абзацы, отмеченные таким значком, предоставляют самую актуальную информацию о концепции, обсуждаемой в текущий момент, или указывают на некий неочевидный факт, который стоит запомнить, — имейте это в виду!

О коде

Во всей книге используется приложение *ServiceMonitor* для демонстрации поведения и особенностей системы модулей. Его можно найти по ссылке github.com/CodeFX-org/demo-jpms-monitor.

С небольшими различиями приложение используется почти во всех главах. В Git-репозитории расположено несколько веток, которые четко демонстрируют особенности, описанные в части I (в основном `master` и некоторые из веток `break-…`) и части III (те или иные ветки `feature-…` и `break-…`).

Часть II, посвященная проблемам с миграцией и модуляризацией, также иногда задействует *ServiceMonitor* как пример, но для нее нет отдельной ветки. Еще один

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

вариант приложения показывает пару других проблем с миграцией: https://github.com/CodeFX-org/demo-java-9-migration.

Все, что нужно для написания кода по мере чтения книги или экспериментов с примерами, — Java версии 9 или выше (см. следующий подраздел), текстовый редактор и минимальные знания командной строки. Если вы решили использовать среду разработки, то выбирайте те, что поддерживают Java 9 (*как минимум* IntelliJ IDEA 2017.2, Eclipse Oxygen.1a или NetBeans 9). Еще я рекомендую набирать команды вручную или запускать сценарии `.sh`- или `.bat`, однако в ряде случаев вам может пригодиться Maven — для создания проектов необходима как минимум версия 3.5.0.

Больше подробностей по каждому из проектов можно найти в их файле `README`.

О версии Java

Java EE становится Jakarta EE

Система модулей — часть *Java Standard Edition 9 (Java SE 9)*. Помимо Java SE, есть еще Java EE — *Java Enterprise Edition (Java EE)*; его текущая версия — Java EE 8. Когда-то Java SE и EE управлялись одним и тем же процессом под крылом одного и того же хранителя — сначала Sun, затем Oracle.

Все изменилось в 2017 году. Oracle передала технологии Java EE в Eclipse Foundation, который основал проект *Eclipse Enterprise для Java (EE4J)* в целях управления ими. Платформа Java EE отныне будет называться *Jakarta EE*, а ее первый выпуск — Jakarta EE 8.

Периодически в данной книге я буду ссылаться на Java EE и Jakarta EE, особенно в разделе 6.1. Во избежание путаницы в названиях двух проектов и вследствие того, что технология все еще формально называется Java EE (или уже Jakarta EE), я буду использовать аббревиатуру *JEE*.

На момент написания этой книги версия Java 9 активно использовалась и весь код гарантированно работал в ней — если быть более точным, в версии 9.0.4. Он также был протестирован и обновлен для версий Java 10 и 11. Когда книга отправилась в печать, версия 11 еще была в раннем доступе, и, возможно, сейчас в ней есть паратройка изменений, на которые нет ссылок в данной книге.

Java 9 не только представил систему модулей — он также стал начальной точкой шестимесячного цикла выпуска. Уже сейчас вышли Java 10 и 11 и даже Java 12 на подходе (вполне вероятно, когда вы читаете это, 12-ю версию уже выпустили). Означает ли вышесказанное, что книга устарела?

К счастью, не совсем. Кроме пары небольших деталей, Java 10 и 11 ничего не изменили в системе модулей; и, если заглядывать в будущее, существенных изменений пока не запланировано. Поэтому, несмотря на то что здесь упоминается только Java 9, книга подходит и для Java 10 и 11, и, скорее всего, для нескольких будущих выпусков.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Это особенно касается проблем совместимости, описанных в части II. У вас не получится списать их со счетов, перепрыгнув на 8-ю или 10-ю версию. К тому же после освоения Java 9 другие версии покажутся гораздо более простыми, поскольку 10-я и 11-я содержат только небольшие изменения без проблем совместимости.

Оформление листингов

Книга содержит множество примеров исходного кода как в пронумерованных листингах, так и вперемешку с обычным текстом. В обоих случаях исходный код оформлен моноширинным шрифтом, подобно этому, чтобы не смешивать его с остальным текстом. (Хотя названия модулей выделены *курсивом* — см. ниже.)

Во многих случаях оригинальный исходный код и выходные данные компилятора или виртуальной машины были переформатированы, чтобы поместиться на странице книги:

- добавлены разрывы строк и переработаны отступы;
- урезаны выходные данные, в частности, убраны названия пакетов;
- сокращены сообщения об ошибках.

В редких случаях даже этого было недостаточно, и потому листинги включают в себя маркеры продолжения строки (➥). Вдобавок комментарии к исходному коду часто убраны из листинга, если код описан в тексте книги. Многие листинги сопровождаются примечаниями, чтобы подчеркнуть важные концепции.

Начиная с Java 8, для ссылки на метод класса часто используется синтаксис ссылки на метод (например, метод `add` из класса `List` обозначается как `List::add`). В отличие от `List.add` он не похож на обычный вызов метода (хотя куда делись круглые скобки?) и не рождает вопросов о количестве параметров. Фактически `List::add` обращается ко всем перегрузкам метода `add`, а не к одной из них. Я использую этот синтаксис на протяжении всей книги.

Оформление имен методов

ВАЖНАЯ ИНФОРМАЦИЯ

Имена модулей и пакетов почти одинаковы по длине, вследствие чего фрагменты кода и диаграммы могут увеличиться. Я отказался от этого, так что все мои собственноручно созданные модули имеют короткие имена. Не поступайте так в реальных проектах! Вместо этого обратитесь к руководству в подразделе 3.1.3 «Декларация модулей: определение свойств модулей».

Поскольку имена пакетов и модулей весьма похожи, я решил выделить имена модулей *курсивом*, а имена пакетов — моноширинным шрифтом. Это позволяет с легкостью различать их, и я призываю вас делать то же самое, если вы пишете о модулях.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Заместители во фрагментах кода

Новые возможности, к примеру флаги командной строки и содержание `module-info.java`, определяются в общих чертах. Поэтому в книге используются `${`*`заместители`*`}`, чтобы выделить специфические значения. Их легко распознать по знаку доллара с последующими фигурными скобками.

Этот синтаксис применяется исключительно в данном контексте, и его сходство с тем, как некоторые операционные системы и языки программирования обращаются к аргументам и переменным, не случайно. Однако он никогда не указывает на какой-либо определенный рабочий механизм, и такие заместители не должны использоваться в операционной системе или JVM сами по себе. Вам придется заполнить их самостоятельно, но обычно где-то рядом в тексте можно найти пояснение, что именно поместить в `${`*`заместитель`*`}`.

Пример. Из подраздела 4.5.3: *«Когда* `jar` *используется для упаковки файлов классов в архив, можно определить основной класс с помощью* `--main-class ${`*`класс`*`}`*, где* `${`*`класс`*`}` *— полное имя (то есть имя пакета, за которым следуют точка и название класса) класса с методом* `main`*».*

Не сложно, правда?

Команды и их вывод

Лучший способ понять систему модулей — использовать ее напрямую, вызывая команды `javac`, `java` и т. д. и читая сообщения, которые Java выведет в командной строке. В данной книге содержится множество переходов от ввода к выводу, от команд к сообщениям. В фрагментах кода перед командами всегда ставится префикс `$`, перед сообщениями — `>`, перед моими комментариями — `#`.

Пример. Вот команда из подраздела 5.3.2:

```
$ java
   --module-path mods
   --validate-modules

# урезанные стандартизированные модули Java
# урезанные нестандартизированные модули JDK
> file:.../monitor.rest.jar monitor.rest
> file:.../monitor.observer.beta.jar monitor.observer.beta
```

Дискуссионный форум liveBook

Приобретая «Систему модулей Java», вы одновременно получаете доступ к приватному форуму от издательства Manning Publications, где можете оставить отзыв о книге, задать технический вопрос и получить помощь от автора или других пользователей. Чтобы попасть на форум, перейдите по ссылке livebook.manning.com/#!/book/the-java-module-system/discussion. Чтобы узнать больше о форумах Manning и их правилах, откройте livebook.manning.com/#!/discussion.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Обязательство издательства Manning по отношению к читателям состоит в обеспечении площадки, где между автором и отдельными читателями может состояться содержательный диалог. Это не значит, что участие автора в форуме является обязательным, — его вклад остается добровольным (и неоплачиваемым). Мы советуем вам задавать автору интересные и сложные вопросы, чтобы его интерес не угас!

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Об авторе

Николай Парлог — 30-летний парень (как выразился бы рассказчик, прищурясь), который нашел свое призвание в разработке программного обеспечения. Он постоянно читает, думает и пишет об этом, а также создает код не только ради денег, но и в свое удовольствие.

Профессионально программируя на Java с 2011 года, Николай впоследствии стал внештатным разработчиком, консультантом и спонсором нескольких проектов с открытым исходным кодом. Он также занимается блогом и новостными рассылками, выступает, записывает видео и общается в чатах — и все это на тему ПО (не одновременно, но весьма эффективно — с высокой пропускной способностью, хотя и с задержками). Его дом — codefx.org, где можно найти ссылки на все его проекты.

Ещё больше книг по Java в нашем телеграм канале:
https://t.me/javalib

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Об обложке

Иллюстрация на обложке книги называется «Житель Флориды» и изображает коренного американца из Флориды. Она взята из коллекции нарядных костюмов разных стран из французской книги *Costumes civils actuels de tous les peuples connus* Жака Грассе де Сен-Совера (Jacques Grasset de Saint-Sauveur) (1775–1810), опубликованной в 1788 году. Каждая иллюстрация создана и раскрашена вручную. Богатство коллекции Грассе де Сен-Совера напоминает о том, как в культурном плане взаимно далеки были города и поселения мира всего 200 лет назад. Изолированные друг от друга, люди говорили на разных языках и диалектах. На улицах города или в селах — везде легко было определить, основываясь лишь на одежде, откуда человек родом, каковы его профессия и материальное положение.

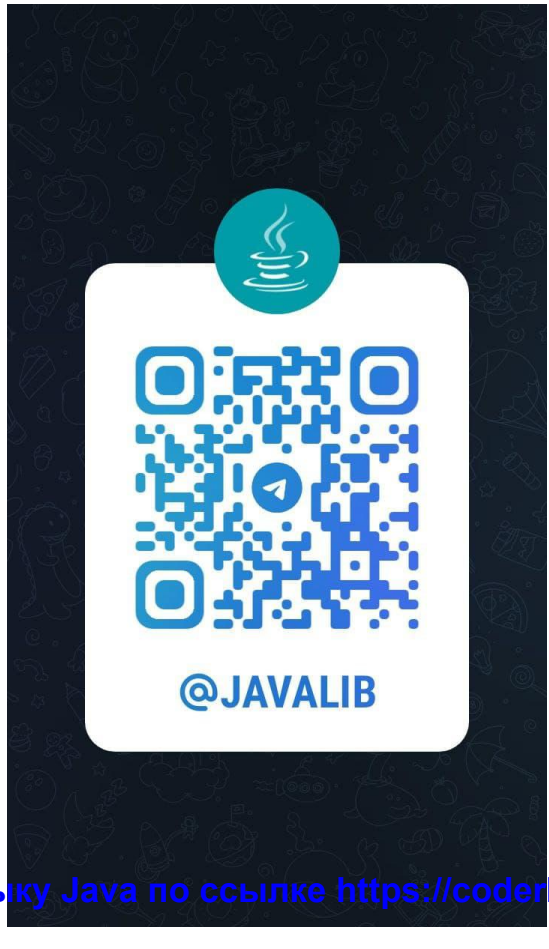
С тех пор манера одеваться значительно изменилась и разнообразие регионов, столь богатое в прошлом, сгладилось. Сегодня сложно различить даже жителей разных континентов, не говоря уже о городах, регионах или странах. Возможно, мы обменяли культурное разнообразие на жизнь, более богатую в личном плане и, конечно, быстро развивающуюся в технологическом.

Сейчас, когда одну техническую книгу сложно отличить от другой, Manning является образцом оригинальности и изобретательности в компьютерном бизнесе, выпуская книги с обложками, которые представляют богатое разнообразие жизни в регионах, имевшееся более двух столетий назад и возрожденное на картинах Грассе де Сен-Совера.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Часть I

Привет, модули



Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Java 9 считает модули первоклассной концепцией. Но что *означают* модули? Какие проблемы они решают и какую выгоду можно из них извлечь? И что же подразумевается под словом *«первоклассный»*?

Книга, которую вы читаете, отвечает на все эти вопросы и на множество других. Она учит, как объявить, спроектировать и запустить модули, а также тому, как они повлияют на проект и какие плюсы привнесут.

Но не все сразу. Эта часть книги начинается с объяснения, что же такое *модульность*, для чего она необходима и какие цели преследует система модулей (глава 1). Глава 2 сразу бросает вас в бой и показывает примеры кода, объявляющего, проектирующего и запускающего модули, после чего главы 3–5 объясняют три этих шага более подробно. Глава 3 особенно важна, поскольку в ней описываются базовые концепции и механизмы системы модулей.

Часть II книги обсуждает проблемы, с которыми Java 9 сталкивается в существующих приложениях, а часть III представляет отдельные расширенные функции.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

1 Первый элемент головоломки

В этой главе

- Как модульность формирует систему.
- Почему Java не может обеспечить модульность.
- Как новая система модулей стремится исправить существующие недостатки.

Всем знакома ситуация, когда развертываемое приложение отказывается работать как положено. На это есть множество причин, но один класс проблем настолько отвратителен, что для него даже придумали слишком милостивое прозвище: *JAR-ад*. Классические его аспекты — неправильные зависимости: одни могут и вовсе отсутствовать, а другие, словно компенсируя данное обстоятельство, появляются несколько раз, и даже в разных версиях. Это верный способ аварийно завершить работу или, что еще хуже, слегка повредить уже запущенные приложения.

Основная проблема, лежащая в основе JAR-ада, заключается в том, что мы рассматриваем JAR как самостоятельные артефакты с идентифицируемостью и зависимостями, в то время как Java видит в JAR лишь простые файловые контейнеры без каких-либо значимых свойств. Подобная разница и приводит к неприятностям.

Первый пример — недостаток значимой инкапсуляции вокруг JAR: все публичные типы видны без ограничений всему коду в приложении. Это позволяет с легкостью и без умысла пользоваться теми типами из библиотек, которые их разработчики считали только деталями реализации и никогда не отшлифовывали для публичного применения. Скорее всего, отладчики скрыли данные типы в пакетах наподобие `internal` или `impl`, но это никак не мешает импортировать их отовсюду.

Затем, когда разработчики изменяют эти внутренние типы, наш код ломается. Или, если у нас достаточно влияния на сообщество данной библиотеки,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

разработчикам приходится оставлять внутренний код нетронутым, что приводит к отсутствию рефакторинга и эволюции кода. Недостаток инкапсуляции приводит к снижению удобства в сопровождении — как для библиотек, так и для приложений.

Менее актуальным для повседневной разработки, но значительно более плохим для экосистемы в целом является то, что критический для безопасности приложения код становится неуправляемым. В Java Development Kit (JDK) это привело к нескольким уязвимостям, которые поспособствовали задержке выхода Java 8 после того, как Oracle выкупила Sun.

Эти и другие проблемы занимали Java-разработчиков более 20 лет, как и обсуждения возможных решений. Java 9 стал первой версией, предоставившей встроенное в язык решение: систему модулей платформы Java (JPMS), разработанную в 2008 году под руководством Project Jigsaw. Она позволяет программистам создавать модули путем добавления метаданных к JAR-файлам, делая последние чем-то большим, чем простые контейнеры. Начиная с Java 9, компилятор и среда выполнения понимают как идентифицируемость модулей, так и связи между ними и потому могут решить проблемы, связанные с дублирующимися зависимостями и недостатком инкапсуляции.

Но система модулей Java не просто заплатка. Она появилась вместе с множеством расширенных функций, которые можно использовать, чтобы написать прекрасное, удобное в сопровождении ПО. Возможно, самое большое преимущество заключается в том, что и разработчик, и все сообщество теперь сталкиваются лицом к лицу с основной концепцией модульности. Больше осведомленных разработчиков, больше модульных библиотек, улучшенная поддержка инструментов — все это можно ожидать от мира, в котором модульность стала почетным гражданином.

Я подозреваю, что многие разработчики просто пропустят несколько прошлых версий Java при обновлении. К примеру, чаще всего Java 8 обновляют сразу до Java 11. Я буду обращать ваше внимание на различия между Java 9, 10 или 11 там, где они появятся. Большинство материалов книги одинаковы для всех версий Java, начиная с 9-й. В некоторых случаях я буду использовать обозначение Java+ как сокращение для «Java версии 9 или выше».

Эта глава начинается с раздела 1.1, где объясняется, что такое модульность и то, как мы обычно воспринимаем структуру программной системы. Суть вот в чем: на определенном уровне абстракции (JAR) виртуальная машина Java видит сущности не так, как мы (см. раздел 1.2). Она просто удаляет тщательно созданную нами структуру! Подобное несоответствие вызывает настоящие проблемы, которые мы обсудим в разделе 1.3. Система модулей была создана в целях превращения артефактов в модули (см. раздел 1.4) и решения проблем, возникающих из-за несоответствия (см. раздел 1.5).

1.1. Что такое модульность

Как вы представляете себе программное обеспечение? Строки кода? Биты и байты? UML-диаграммы? Файлы POM в Maven?

Я хочу, чтобы вы опирались не на определение, а на ощущение. Остановитесь и подумайте о своем любимом проекте (или хотя бы о том, за который вам заплатили). На что он похож? Как вы его визуализируете?

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

1.1.1. Визуализация ПО в виде графов

Я вижу кодовые базы как систему взаимодействующих частей. (Да-да, *настолько* формально.) Каждая часть имеет три основные составляющие: имя, зависимости от остальных частей и особенности, которые она предоставляет другим частям.

Данное представление справедливо для всех уровней абстракции. На очень низком уровне часть — отдельный метод, где ее имя — это имя метода, ее зависимости — методы, вызываемые им, а ее особенности — это его возвращаемое значение или изменение состояния, на которое он повлиял. На очень высоком уровне часть соотносится с сервисом (кто-то сказал «микро»?) или даже со всем приложением.

Представьте сервис оформления заказов: он, как часть интернет-магазина, позволяет пользователям покупать выбранные товары. Чтобы сделать это, ему нужно вызвать сервисы авторизации и платежных карт. Мы снова получили три свойства: имя, зависимости и особенности. Данную информацию можно с легкостью применить для построения схемы (рис. 1.1).

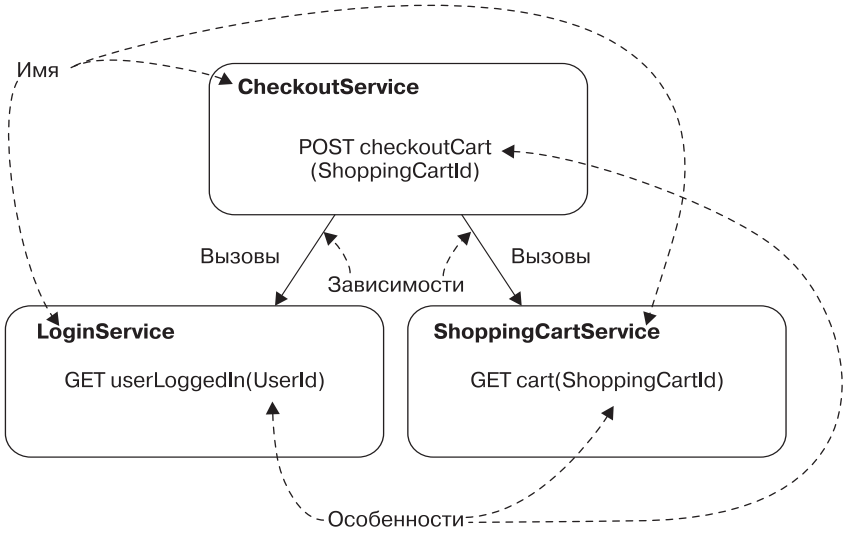


Рис. 1.1. Если изобразить сервис оформления заказов и его зависимости, то они естественным образом сформируются в виде небольшого графа, показывающего их имена, зависимости и особенности

Мы можем рассматривать части на разных уровнях абстракции. Между такими крайностями, как методы и приложения, существуют еще классы, пакеты и JAR-файлы. У них тоже есть имена, зависимости и особенности.

Что самое интересное — мы можем использовать эти сущности для визуализации и анализа системы. Если представить или даже нарисовать узел для каждой части и затем соединить его с другими в соответствии с зависимостями, то получится граф.

Это происходит настолько естественно, что для примера с интернет-магазином все уже готово — а вы, вероятно, и не заметили. Взгляните на другие способы визуализировать программные системы, как на рис. 1.2, — графы появляются везде.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Рис. 1.2. В разработке ПО графы вездесущи. Они бывают разных видов и форм: к примеру, UML-диаграммы *(слева)*, деревья зависимостей Maven *(в центре)* и графы связности микросервисов *(справа)*

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Диаграммы классов — это графы. Зависимости в выходных данных инструментов сборки структурированы в виде деревьев (если вы используете Gradle или Maven, то попробуйте ввести `gradle dependencies` или `mvn dependency:tree` соответственно), которые являются отдельным видом графа. Вам встречались запутанные диаграммы микросервисов, где ничего нельзя понять? Это тоже графы.

Графы выглядят по-разному в зависимости от того, говорим ли мы о компиляции или выполнении, рассматриваем ли один уровень абстракции или их набор, оцениваем весь жизненный цикл системы или определенный момент времени, а также в зависимости от других возможных различий. Некоторые отличительные признаки станут важными позже, и в настоящее время нет необходимости рассматривать их подробно. Сейчас же подойдет любой из типов графов — просто представьте наиболее удобный для вас.

1.1.2. Влияние принципов проектирования

Визуализация системы в виде графа — это самый распространенный способ оценить ее архитектуру. Многие из принципов проектирования хорошего программного обеспечения влияют на то, как оно выглядит.

Возьмем для примера принцип *разделения ответственности*. Следуя ему, мы должны стремиться создать ПО, где каждая часть фокусируется на отдельной задаче (наподобие «войти в систему» или «нарисовать карту»). Часто задачи состоят из небольших задач (например, «загрузить пользователя» и «проверить пароль» для входа в систему), и части, которые этим занимаются, тоже должны быть отделены друг от друга. Подобное положение приводит к появлению графа, где отдельные части образуют небольшие кластеры, реализующие четко определенные задачи.

И наоборот: если плохо разделить ответственность, то граф не будет иметь четкой структуры и станет выглядеть так, словно каждый узел в нем соединяется со всеми остальными. Как вы можете увидеть на рис. 1.3, эти два случая легко распознать.

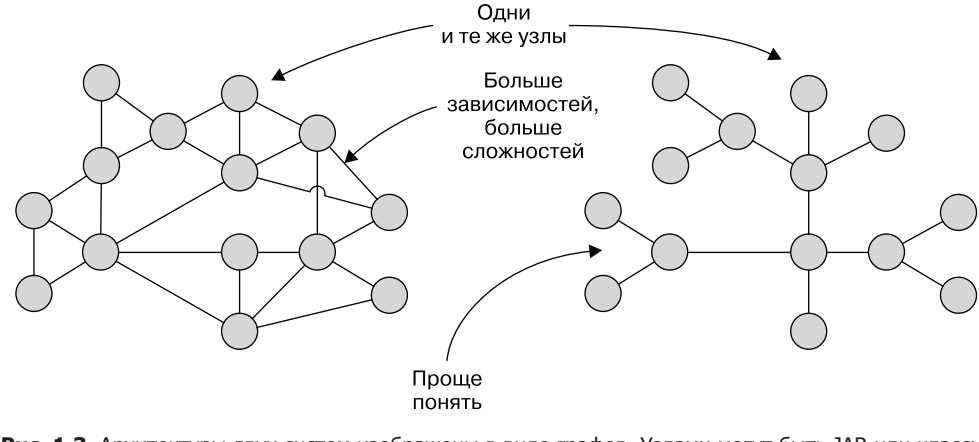


Рис. 1.3. Архитектуры двух систем изображены в виде графов. Узлами могут быть JAR или классы, а линиями — связи между ними. Но детали не так важны: все что нужно — быстро взглянуть и ответить на вопрос, представлено ли здесь правильное разделение ответственности

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Другой пример принципа, влияющего на вид графа, — принцип *инверсии зависимостей*. Во время выполнения высокоуровневый код всегда обращается к низкоуровневому, но правильно спроектированная система инвертирует данные зависимости во время компиляции: высокоуровневый код зависит от интерфейсов, а низкоуровневый применяет их, таким образом инвертируя эти зависимости вверх к интерфейсам. Взглянув на правильный вариант графа (рис. 1.4), можно легко различить эти инверсии.

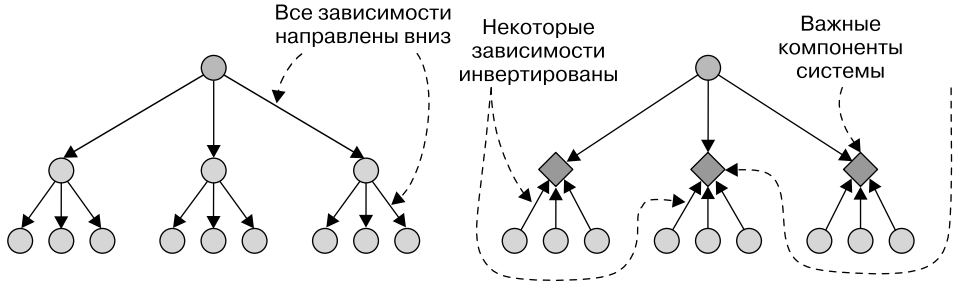


Рис. 1.4. Система, в которой высокоуровневый код опирается на низкоуровневый, образует граф *(слева)*, отличный от использующего интерфейсы для инвертирования зависимостей вверх *(справа)*. Инверсия позволяет с легкостью идентифицировать и понять значимые компоненты системы

Цель таких принципов, как разделение ответственности и инверсия зависимостей, — распутать граф. Если проигнорировать их, то система превратится в хаос, где ничто не может быть изменено без потенциального вывода из строя чего-то никак с этим не связанного. Если же следовать им — система станет хорошо организованной.

1.1.3. Что такое модульность

Принципы хорошего проектирования ПО позволяют создавать простые системы. Несмотря на то, что конечная цель — удобные в сопровождении *системы*, большинство принципов направлены на работу с отдельными *частями*. Принципы фокусируются не на всей кодовой базе, а на отдельных элементах, поскольку в итоге их свойства определяют свойства собранной из них системы.

Мы уже рассмотрели то, какие преимущества предоставляют разделение ответственности и инверсия зависимостей: фокус на отдельных задачах и зависимость от интерфейсов, а не от способов реализации. Это и есть самые важные особенности частей системы.

ВАЖНАЯ ИНФОРМАЦИЯ

Каждый модуль, который я называл до этого частью, имеет четкие обязанности и соглашения для реализации. Он автономен, непрозрачен для своих клиентов и может быть заменен другим модулем, если тот выполняет те же функции. Он немного зависит от API, но не от реализации.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Системы, построенные из подобных модулей, лучше поддаются изменениям, с учетом реализации зависимостей более гибки при запуске и, возможно, даже во время выполнения. И в этом заслуга модульности — она делает хорошо спроектированные модули гибкими и удобными в сопровождении.

1.2. Уничтожение модулей до Java 9

Вы уже заметили, как граф взаимосвязанных частей соединяется с парой хороших свойств, в чем и заключается модульность. Но это только идеи и повод поговорить о разработке ПО. Граф — всего лишь строки кода, которые в случае с Java компилируются в байт-кодовые инструкции и запускаются с помощью виртуальной машины Java (JVM). Было бы замечательно, если бы язык, компилятор и JVM (все это я грубо и некорректно обобщу термином *Java*) могли видеть сущности так же, как их видим мы.

Чаще всего они это и делают! При проектировании класса или интерфейса имя, которое вы ему присвоите, станет идентификатором и для Java. Определенные вами методы — те, что могут быть вызваны другим кодом, — будут для Java теми же методами, с такими же именами и параметрами. Их зависимости ясно видны как в выражениях импорта, так и в полных именах классов, и компилятор с JVM станут использовать классы с этими именами для выполнения.

В качестве примера рассмотрим интерфейс `Future`, представляющий собой результат вычисления, которое может быть (или нет) завершено. Функциональность типа неважна, поскольку нас интересуют лишь его зависимости:

```
public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);
boolean isCancelled();
boolean isDone();
V get() throws InterruptedException, ExecutionException;
V get(long timeout, TimeUnit unit)
  throws InterruptedException,
    ExecutionException,
    TimeoutException;
}
```

Просматривая методы, объявленные в `Future`, можно легко перечислить зависимости:

- `InterruptedException`;
- `ExecutionException`;
- `TimeUnit`;
- `TimeoutException`.

Применяя тот же анализ к только что определенным типам, создадим граф зависимостей, как на рис. 1.5. Его точная форма не имеет значения. Гораздо важнее то, что граф, который существует у нас в голове, когда мы говорим о типе, в точности совпадает с графом, явно созданным Java.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

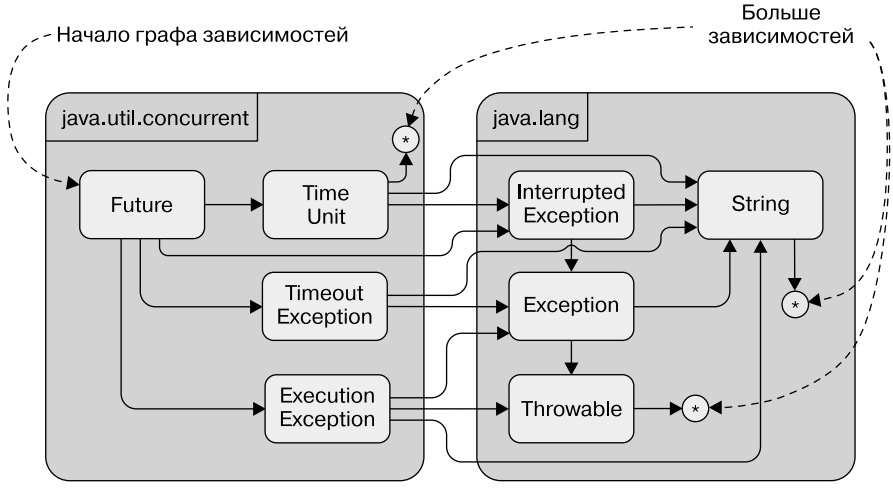


Рис. 1.5. Граф зависимостей, с которым работает Java для любого из предложенных типов, совпадает с нашим восприятием зависимостей этого типа. Данный граф показывает зависимости интерфейса Future между пакетами java.util.concurrent и java.lang

Поскольку Java сильно и статически типизирован, вы сразу получите сообщение, если что-то даст сбой. Неправильное имя класса? Одна из зависимостей утеряна? Видимость метода была изменена и теперь другие методы не видят его? Java уведомит обо всем этом компилятор во время сборки и JVM во время выполнения.

Проверки во время компиляции позволяет обойти рефлексия (кратко об этом — в приложении Б). По данной причине она считается грубым и потенциально опасным инструментом, который можно использовать только в исключительных случаях. Пока мы ее проигнорируем, но вернемся к ней в следующих главах.

В качестве примера того, как расходятся наше восприятие зависимостей и восприятие Java, рассмотрим уровень сервиса или приложения. Это выходит за рамки Java: он не может знать об используемом приложении, как не в состоянии и предупредить о том, что не имеет доступа к сервису GitHub или базе данных Oracle (упс!), и также не знает о поломке со стороны клиента, вызванной вашими изменениями в API. Он не может составить карту совместной работы приложений или сервисов. И это нормально, поскольку Java работает на уровне *отдельных* приложений.

Но один уровень абстракции явно находится в сфере влияния Java, невзирая на крайне слабую поддержку до Java 9 — настолько слабую, что усилия по модуляризации фактически были напрасны, приведя к тому, что называется *уничтожением модулей*. Этот уровень имеет дело с артефактами, или *JAR* — на языке Java.

Если приложение на этом уровне модуляризовано, то оно состоит из нескольких JAR, а если нет — зависит от библиотек, которые могут иметь свои собственные зависимости. Отобразив их, вы получите уже знакомый граф, но на этот раз для JAR, а не для классов.

В качестве примера рассмотрим приложение под названием *ServiceMonitor*. Если не слишком вдаваться в подробности, то оно работает следующим образом: проверя-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ет доступность других сервисов и собирает статистику. Все это записывается в базу данных и может быть получено через REST API.

Авторы приложения создали четыре JAR:

- *observer* — следит за сервисами и проверяет доступность;
- *statistics* — составляет статистику на основе доступных данных;
- *persistence* — читает и записывает статистику в базу данных с *hibernate*;
- *monitor* — запускает сбор данных и направляет их через *statistics* в *persistence*, реализует REST API со *spark*.

Каждый JAR имеет собственные зависимости, каждая из которых представлена на рис. 1.6.

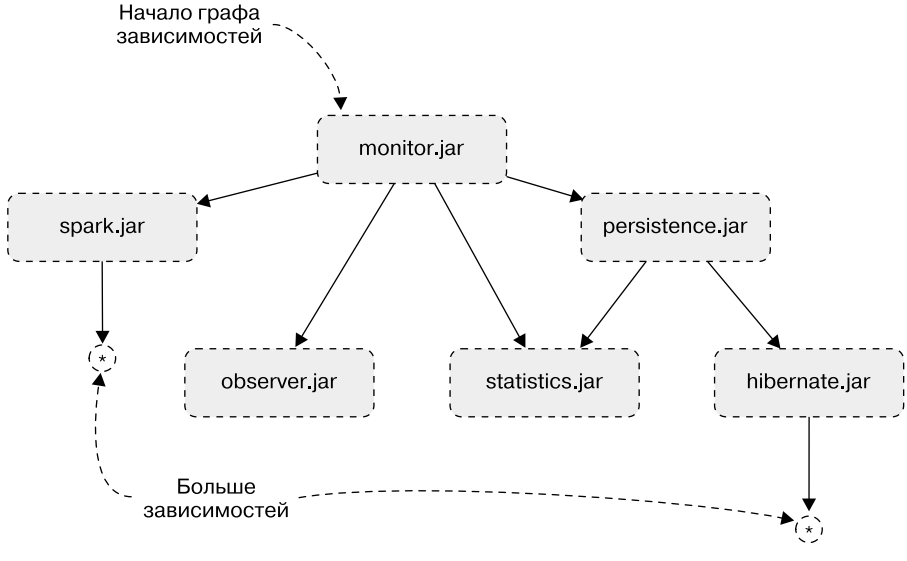


Рис. 1.6. Для любого приложения можно нарисовать граф зависимостей между его артефактами. Здесь представлено приложение ServiceMonitor, разбитое на JAR, которые имеют зависимости не только друг от друга, но и от сторонних библиотек

Граф включает все, что было описано ранее: у всех JAR есть имена, зависимости друг от друга и специфические особенности, позволяющие другим JAR использовать публичные методы и классы.

При запуске приложения вы должны обязательно перечислить пути классов и JAR, которые планируете применять:

```
$ java
--class-path observer.jar:statistics.jar:persistence.jar:monitor.jar
org.codefx.monitor.Monitor
```

Явно перечисляйте необходимые JAR-файлы с параметром --class-path (новый вариант для -cp и -classpath, который также работает с javac)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

И вот здесь все идет не так — по крайней мере, до Java 9. JVM запускается, ничего не зная о ваших классах. Каждый раз, когда она встречает обращение к неизвестному классу, которое начинается с главного класса, указанного в командной строке, она просматривает все JAR в пути класса в поисках класса с подходящим именем. Если таковой найден, то JVM запускает его со всеми остальными классами и завершает работу. Как видите, в JVM нет концепции среды выполнения, соответствующей JAR-файлам.

Без представления в среде выполнения JAR теряют свою идентифицируемость. Хоть у них и есть собственные имена, для JVM это не имеет значения. Разве не было бы хорошо, если бы сообщения об ошибках могли указывать на JAR, в котором возникла проблема, или если бы JVM могла назвать отсутствующую зависимость?

К слову, о зависимостях — они тоже становятся невидимыми. Работая на уровне классов, JVM не имеет представления о зависимостях между JAR-файлами. Игнорирование артефактов, содержащих классы, делает невозможным инкапсуляцию этих артефактов. И действительно, каждый публичный класс виден всем остальным классам.

Имена, явные зависимости, четко объявленные API — ни компилятор, ни JVM не интересуются тем, что мы так ценим в модулях. Это уничтожает модульную структуру и превращает аккуратно спроектированный граф в комок грязи, как показано на рис. 1.7. И такое не проходит бесследно.

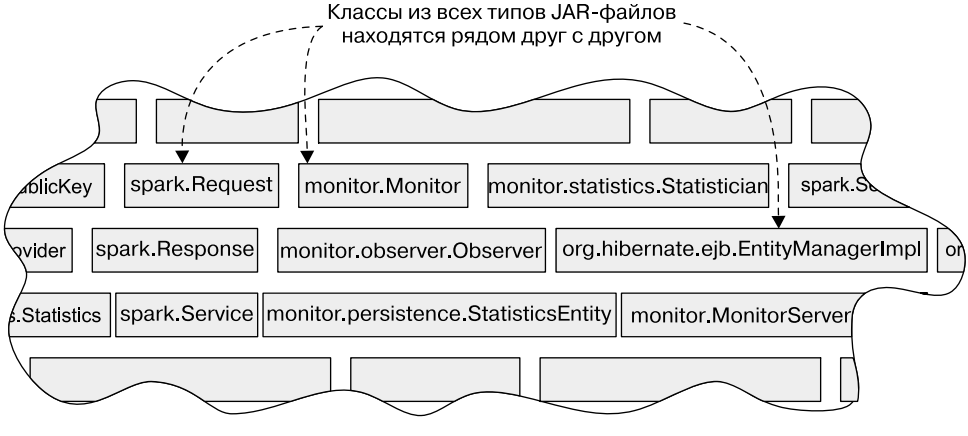


Рис. 1.7. Ни Java-компилятор, ни его виртуальная машина не имеют представления об артефактах и зависимостях между ними. Несмотря на то что JAR относятся к простым контейнерам, вне классов они загружаются в отдельное пространство имен. В итоге классы превращаются в первичный бульон, где каждый публичный тип доступен любому другому

1.3. Трудности до Java 9

Как вы могли заметить, в Java до версии 9 отсутствовали концепции для правильной поддержки модульности между артефактами. И хотя это вызывало проблемы, их,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

очевидно, можно было решить (иначе мы бы не использовали Java). Но, когда наконец они проявлялись, обычно в более крупных приложениях, с ними практически нельзя было справиться.

Как я уже упоминал в начале главы, сложности, с максимальной вероятностью влияющие на процесс разработки, объединены под общим названием *«JAR-ад»*; но есть и другие. Проблемы сопровождения и безопасности, в большей степени актуальные для разработчиков JDK и библиотек, тоже имеют последствия.

Я уверен, что вы также встречались с этими трудностями, и на протяжении следующей главы мы подробно рассмотрим их одну за другой. Не переживайте, если они вам не знакомы, — вам просто повезло с ними не столкнуться. Если же вам знакомы JAR-ад и связанные с ним проблемы, то можете сразу перейти к разделу 1.4, который представит систему модулей.

Если вы разочарованы этим, по видимости, бесконечным потоком неполадок, то расслабьтесь — за ним последует возмездие: раздел 1.5 расскажет, как модули помогут справиться с большинством подобных проблем.

1.3.1. Невыраженные зависимости между JAR-файлами

Было ли у вас приложение, которое «падало» с ошибкой *NoClassDefFoundError*? Это происходило потому, что JVM не могла найти класс, ссылка на который есть в выполняемом коде. Найти зависимый код легко (достаточно просмотреть трассировку стека, чтобы выявить его), и идентификация потерянной зависимости тоже не требует много работы (ее выдает отсутствующее имя класса). Но понять, *почему* зависимости нет, бывает очень тяжело. Однако, учитывая граф зависимости артефактов, возникает вопрос — почему мы обнаруживаем, что чего-то не хватает, только во время выполнения?

ВАЖНАЯ ИНФОРМАЦИЯ

Причина проста: JAR не может выразить, на какой другой JAR-файл он ссылается так, чтобы это было понятно виртуальной машине. Для идентификации и применения зависимостей требуется внешняя сущность.

До того как инструменты сборки научились распознавать и извлекать зависимости, мы и пользовались этими внешними сущностями. Приходилось сканировать документацию на предмет зависимостей, искать необходимые проекты, загружать JAR-файлы и добавлять их в проект. Дополнительные зависимости, когда одному JAR может потребоваться другой JAR для каких-то специфических возможностей, значительно усложняли задачу.

Чтобы приложение работало, ему может понадобиться сразу несколько библиотек. Но каждая из них может потребовать еще несколько и т. д. По мере усугубления проблемы невыраженных зависимостей проект становится все более громоздким и подверженным ошибкам.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

Инструменты сборки, такие как Maven и Gradle, во многом решили эту проблему. Они преуспели в превращении неявных зависимостей в явные так, чтобы можно было выследить каждый нужный JAR вдоль множества ребер переходного дерева зависимостей. Тем не менее понимание JVM концепции зависимостей между артефактами повысило бы надежность и переносимость.

1.3.2. Затенение классов с одинаковыми именами

Иногда случается так, что разные JAR-файлы в пути класса имеют одинаковые полные имена. Это происходит по целому ряду причин:

- разные версии одной и той же библиотеки;
- JAR может содержать собственные зависимости — это называется «*толстый* JAR» или «*uber*-JAR», — однако некоторые из них собраны как отдельные JAR-файлы, поскольку от них зависят другие артефакты;
- библиотека может быть переименована или разделена, и некоторые ее типы непроизвольно будут добавлены в один и тот же путь класса.

Определение: затенение

Поскольку класс будет загружен из первого JAR-файла в пути класса, который его содержит, все другие классы с такими же именами станут недоступными — это значит, что они *затенены*.

Если варианты классов различаются семантически, то это приведет к чему-либо от «слишком-неявного-чтобы-быть-замеченным» поведения до ошибки, вызывающей хаос и разрушения. Хуже того, форма, в которой проявляется проблема, может оказаться недетерминированной. Все зависит от порядка, в котором были найдены JAR-файлы: он может быть разным в зависимости от используемой среды: к примеру, от применяемой вами среды разработки (IntelliJ, Eclipse или NetBeans) до конечной машины, где код будет непосредственно запускаться.

Возьмем, например, широко используемую компанией Google библиотеку *Guava*, которая содержит служебный класс `com.google.common.collect.Iterators`. При переходе от 19-й версии Guava к 20-й метод `emptyIterator()` был удален. Как показывает рис. 1.8, если обе версии будут находиться в одном и том же пути класса *и* если версия 20 будет применена первой, то есть весь код, зависящий от `Iterators`, станет задействовать новую версию, тогда вызов `Iterators::emptyIterator` из 19-й версии окажется невозможным. Метод становится невидимым, даже несмотря на то, что класс, содержащий его, находится в том же пути класса.

Затенение чаще всего происходит случайно. Но это же поведение может быть использовано нарочно, для перезаписи отдельных классов из сторонних библиотек собственными реализациями с одновременным исправлением библиотеки. Хотя инструменты сборки могут снизить вероятность случайного затенения, обычно им не под силу предотвратить его.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

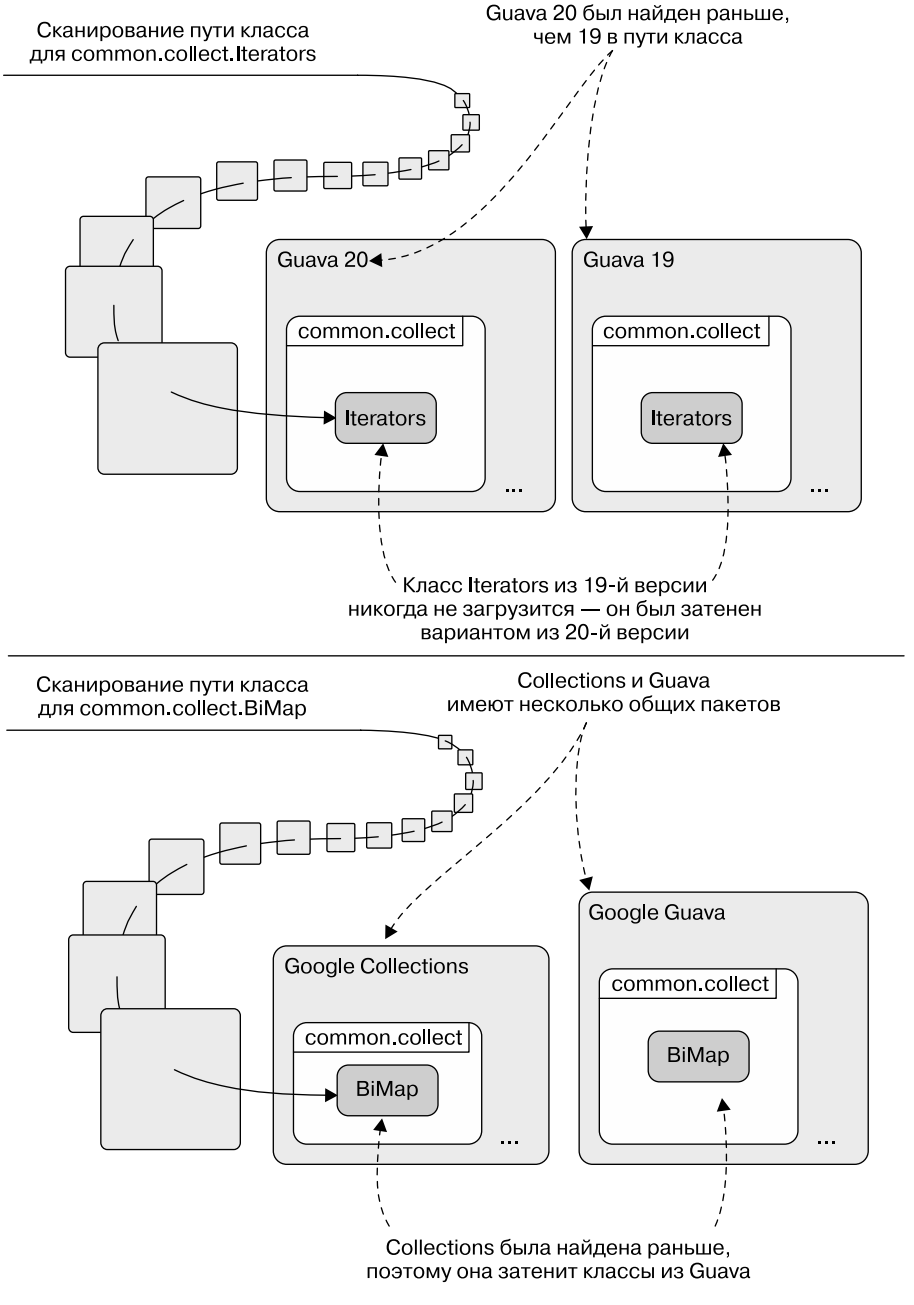


Рис. 1.8. Бывает так, что путь класса содержит разные версии одной и той же библиотеки *(сверху)* или две библиотеки имеют набор общих типов *(снизу)*. В обоих случаях некоторые типы представлены больше одного раза. Во время сканирования пути класса загрузится только первый вариант (затеняя остальные), поэтому лишь порядок сканирования JAR-файлов определяет, какой код будет запущен

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

1.3.3. Конфликты разных версий на одном проекте

Конфликты версий — проклятие любого большого программного проекта. Когда количество зависимостей обозначается больше чем одной цифрой, вероятность возникновения конфликтов стремится к 1 с ужасающей скоростью.

> **Определение: конфликт версий**
>
> *Конфликт версий* возникает, когда две необходимые библиотеки зависят от разных, несовместимых версий сторонней библиотеки.

Если обе версии находятся в одном и том же пути класса, то их поведение становится непредсказуемым. Вследствие затенения классы, которые находятся в обеих версиях, будут загружены лишь из одной из них. Хуже того, даже если класс существует только в одной из версий — он загрузится в любом случае. Вызов кода в библиотеке может обнаружить смесь двух версий.

С другой стороны, если одна из версий пропущена, то программа может работать некорректно, поскольку обе версии обязательны и несовместимы и не могут заменить друг друга (рис. 1.9). Аналогично случаям с потерянными зависимостями ошибка будет определена как «неожиданное поведение», или `NoClassDefFoundError`.

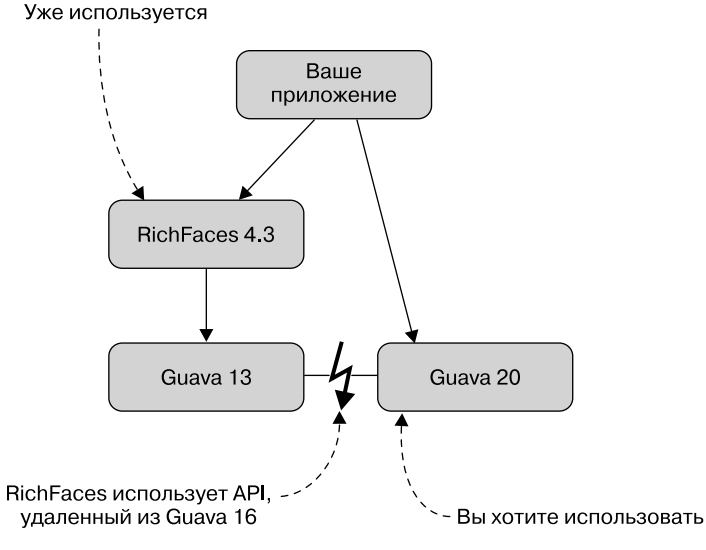


Рис. 1.9. Переходные зависимости конфликтующих версий иногда невозможно разрешить — одна из зависимостей должна быть устранена. Здесь старая версия RichFaces зависит не от той версии Guava, которая используется в приложении. К несчастью, из Guava 16 был удален API, на который ссылался RichFaces

Продолжая рассматривать пример с Guava из раздела о затенении, представьте, что какой-нибудь код ссылается на `com.google.common.io.InputSupplier`, класс,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

который представлен в 19-й версии, но исключен из 20-й. JVM первым делом просканирует Guava 20 и, не найдя класс, загрузит его из Guava 19. Внезапно смесь из двух версий Guava продолжит работу! И в заключение представьте, что `InputSupplier` вызовет `Iterators::emptyIterator`. Как думаете, насколько весело будет отлаживать все это?

ВАЖНАЯ ИНФОРМАЦИЯ

Не существует технического решения этой проблемы, при котором не нужно было бы использовать имеющиеся системы модулей или управлять загрузчиками классов вручную. Инструменты сборки в основном могут определить эти сценарии. Они могут показать предупреждение либо сами разрешат проблему, например, используя последнюю версию.

1.3.4. Сложная загрузка класса

Наше исследование механизма загрузки класса, начавшееся в разделе 1.2, еще не завершено. Описанное поведение справедливо для приложений, использующих один и тот же загрузчик класса. Но разработчики могут свободно добавлять новые загрузчики, делегируя решение проблем, которые мы обсуждали ранее, от одного загрузчика к другому.

Подобное поведение типично для таких контейнеров, как системы компонентов и веб-серверы. В идеале явное его использование должно быть скрыто от разработчиков приложения, но, как известно, ни одна абстракция не умеет хранить секреты. И при некоторых обстоятельствах разработчики могут явно добавить загрузчик класса для реализации определенных свойств: например, чтобы позволить пользователям расширять приложение путем добавления новых классов или чтобы применять конфликтующие версии для одной и той же зависимости.

Независимо от того, как именно появятся множественные загрузчики класса, вам потребуется изучить эту тему как можно тщательнее. Их использование быстро может привести к сложным механизмам делегирования, из-за чего код может повести себя непредсказуемо.

1.3.5. Слабая инкапсуляция в JAR-файлах

Модификаторы доступа Java прекрасно подходят для реализации инкапсуляции в классах внутри одного и того же пакета. Но за пределами пакета существует только один вид доступа для типов: `public`.

Как вы уже заметили, загрузчик класса превращает все загружаемые классы в один *большой комок грязи*[1] — так, что все публичные классы доступны для всех

[1] Большой комок грязи (англ. big ball of mud) — программная система с нераспознаваемой архитектурой. Такие системы распространены на практике из-за давления бизнеса, текучки кадров среди разработчиков и энтропии кода, хотя их использование нежелательно с точки зрения программной инженерии. Они представляют собой пример антишаблона дизайна (https://ru.wikipedia.org/wiki/Большой_комок_грязи). — *Примеч. ред.*

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

остальных. Из-за такой *слабой инкапсуляции* становится невозможным создать функциональность, которая будет видна внутри JAR, но не снаружи.

Все описанное очень затрудняет модуляризацию системы. Если некая функциональность необходима разным частям модуля (например, библиотеке или подпроекту вашей системы), но не должна быть видна извне, единственный способ достичь этого — собрать все в один пакет и пользоваться видимостью внутри него. В результате такого упреждающего повиновения вы сами уничтожаете структуру кода вместо того, чтобы предоставить эту возможность виртуальной машине. Даже в том случае, когда видимость внутри пакета решает проблему, все еще существует рефлексия, позволяющая получить доступ к коду извне.

Слабая инкапсуляция позволяет клиентам артефакта проникать внутрь него (рис. 1.10). Это может произойти случайно, если IDE предлагает импортировать классы, помеченные в документации как внутренние. Чаще всего подобное делается намеренно, чтобы обойти проблемы, для которых не находится другого решения (что иногда верно, а иногда — нет). Однако такой выбор дорого обходится!

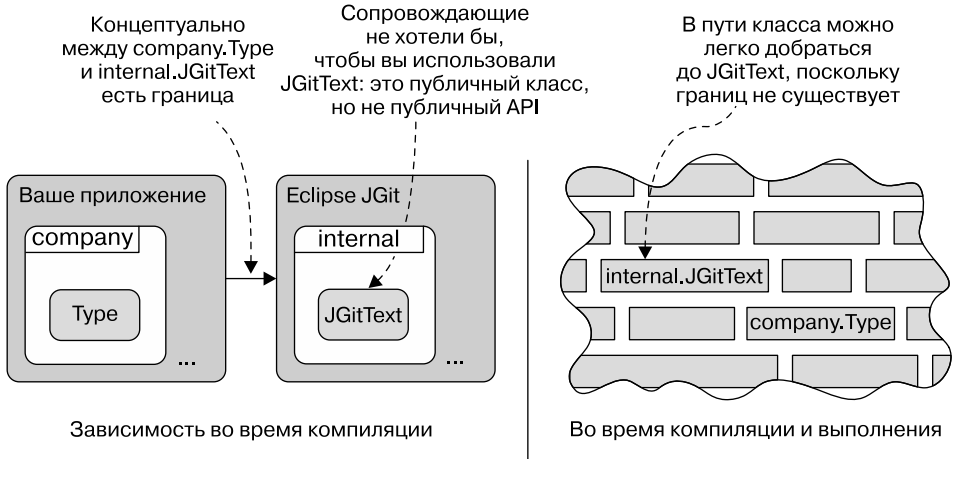


Рис. 1.10. Сопровождающие Eclipse JGit не предусматривали публичное использование org.eclipse.jgit.internal. К несчастью, поскольку Java не имеет понятия о содержимом JAR, сопровождающие никак не могут предотвратить компиляцию любого com.company.Type. Даже если видимость настроена в пределах пакета, это все еще можно обойти с помощью рефлексии

Теперь клиентский код связан с деталями реализации артефакта. Это делает обновления рискованными для клиента и, если сопровождающие решили составить соглашение для данной связности, мешает изменить внутреннюю реализацию. Это может привести к замедлению или даже остановке эволюции артефакта.

Это может показаться исключительным случаем, но это не так. Пресловутый пример — `sun.misc.Unsafe`, внутренний класс JDK, который позволяет делать сумасшедшие (по стандартам Java) вещи, такие как прямое выделение и освобождение памяти. Многие особо важные библиотеки Java, например Netty, PowerMock, Neo4J, Apache Hadoop и Hazelcast, используют его. И поскольку многие приложения опираются на эти библиотеки, они также зависят и от их содержимого. Таким образом, `Unsafe`

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

становится значительной частью инфраструктуры, хотя он не был ни предназначен, ни спроектирован для этого.

Еще один пример — JUnit 4. Во многие инструменты, особенно в IDE, встроены разные удобные возможности, чтобы упростить разработчикам тестирование. Но, поскольку API JUnit 4 недостаточно крут, чтобы все их реализовать, ему приходится вторгаться во внутренний код этих инструментов. Такая связка существенно замедляет разработку JUnit 4 и в конце концов становится причиной перехода на JUnit 5.

1.3.6. Проверки безопасности должны быть выполнены вручную

Моментальное последствие слабой инкапсуляции в пределах пакета проявляется в том, что функционал, отвечающий за безопасность, распределяется на весь код внутри среды разработки. Это значит, что зараженный код может проникнуть в критически важный функционал, и единственный способ противостоять этому — *вручную проверить безопасность* на критическом пути выполнения.

Со времен Java 1.1 это достигалось с помощью вызова `SecurityManager::checkPackageAccess`, который проверял, имеется ли у вызывающего кода доступ к вызываемому пакету — в каждом пути кода, отвечающего за безопасность. Более того, его *нужно* было вызывать в любом таком пути. Забыть о подобном вызове означало сделать код уязвимым для атак, от которых Java страдал в прошлом, особенно во время перехода от Java 7 к 8.

Конечно, можно поспорить с тем, что необходимо дважды, трижды или четырежды проверять код, отвечающий за безопасность. Однако не стоит забывать о человеческом факторе: ручные проверки безопасности на границах модуля представляют более высокий риск, чем хорошо автоматизированный вариант.

1.3.7. Низкая производительность при загрузке

Вы когда-нибудь задумывались, почему многим Java-приложениям, особенно вебсерверам, которые используют мощные фреймворки наподобие Spring, требуется так много времени для загрузки?

> **Определение: ленивая загрузка**
>
> Как вы уже заметили, JVM будет *лениво* загружать классы по мере их надобности. Чаще всего многие классы доступны сразу после запуска (а не через какой-то промежуток времени). И Java требуется немало времени, прежде чем все они будут загружены.

Одна из причин — загрузчик класса не может заранее знать, из какого JAR-файла будет загружен класс, поэтому линейно проходит по всем JAR в пути класса. Аналогично для идентификации всех вхождений конкретной аннотации требуется проверка всех классов в пути класса.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

1.3.8. Негибкая среда выполнения в Java

На самом деле эта проблема не является следствием подхода «большой комок грязи» в JVM, но если мы ее не объявим, то избавиться от нее не получится.

> **Определение: негибкая среда выполнения**
>
> До Java 8 не было никакой возможности установить только выбранные элементы среды выполнения. Все установщики Java включали в себя, например, XML, SQL и Swing, которые во многих случаях являются необязательными.

Хотя это не так существенно для устройств средних размеров (настольные ПК и ноутбуки), очевидно, что это важно для небольших устройств, таких как роутеры, приставки и всевозможные другие приборы, в которых используется Java. Учитывая нынешнюю тенденцию к контейнеризации, это актуально и для серверов, уменьшение площади которых прямо пропорционально влияет на количество расходов на их содержание. Java 8 представила компактные профили, определяющие три подраздела Java SE. Они смягчают проблему, однако не решают ее полностью. Компактные профили фиксированы и потому не могут охватить все текущие и будущие нужды отдельных частей сред выполнения.

1.4. Взгляд на модули с высоты птичьего полета

Мы только что обсудили несколько проблем. Какое их решение предлагает система модулей платформы Java? Основная идея до невозможности проста!

ВАЖНАЯ ИНФОРМАЦИЯ

Модули — это строительные блоки системы модулей платформы Java (неожиданно, правда?). Как и JAR-файлы, они являются контейнерами для типов и ресурсов, но, в отличие от JAR, имеют дополнительные характеристики. Вот самые основные:

- имя, желательно уникальное в глобальном смысле;
- декларация зависимостей от других модулей;
- явно определенный API, который состоит из экспортированных пакетов.

1.4.1. Все на свете — это модули

Существуют различные типы модулей. Они будут рассортированы по категориям в подразделе 3.1.4, но есть смысл взглянуть на них уже сейчас. Во время работы над Project Jigsaw OpenJDK была разделена на более чем 100 модулей, также называемых *платформенными*. Около 30 из них имеют имена, начинающиеся с `java.*`; это стандартизированные модули, они должны быть включены в каждую JVM (на рис. 1.11 показаны некоторые из них).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Это не просто концептуальная схема, а то, как модульная система «видит» сущности

Рис. 1.11. Подборка платформенных модулей. Стрелками показаны их зависимости, однако некоторые из них были убраны, чтобы не загромождать схему: модуль-агрегатор java.se напрямую зависит от каждого модуля и каждый модуль напрямую зависит от java.base

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Ниже представлены еще несколько важных модулей:

- *java.base* — модуль, без которого не будет функционировать ни одна JVM-программа. Содержит такие пакеты, как `java.lang` и `java.util`;
- *java.desktop* — предназначен не только для самых смелых UI-разработчиков программ на ПК. Содержит набор абстрактных окон (AWT, пакеты `java.awt.*`), Swing (пакеты `javax.swing.*`) и другие API, в том числе JavaBeans (включает пакет `java.beans.*`);
- *java.logging* — содержит пакет `java.util.logging`;
- *java.rmi* — внутренний вызов метода (RMI);
- *java.xml* — содержит почти все из набора XML API: Java API for XML Processing (JAXP), Streaming API for XML (StAX), Simple API for XML (SAX), а также объектную модель документа (DOM);
- *java.xml.bind* — Java Architecture for XML Binding (JAXB);
- *java.sql* — Java Database Connectivity (JDBC);
- *java.sql.rowset* — JDBC RowSet API;
- *java.se* — обращается к модулям, составляющим основную часть Java SE API (это так называемый *модуль-агрегатор*, см. подраздел 11.1.5);
- *java.se.ee* — обращается к модулям, составляющим полный Java SE API (еще один агрегатор).

За этим следует JavaFX. Стопроцентный знак превосходства его высокоуровневой архитектуры над AWT и Swing — не только то, что он был вычленен из остатков JDK в отдельный модуль, но и то, что и сам он впоследствии был разбит еще на семь модулей: связи, графика, управление, веб-отображение, FXML, медиа и Swing interop. Имена всех этих модулей начинаются с *javafx.**.

И в заключение, есть еще около 60 модулей, имена которых начинаются с *jdk*. Они содержат реализации API, внутренние утилиты, инструменты (такие как компилятор, JAR, Java Dependency Analysis Tool (JDeps) и Java Shell Tool (JShell)) и др. Они могут различаться в зависимости от реализации JVM, поэтому их использование сродни применению кода из пакетов `sun.`: это выбор не на будущее, а на случай, когда ничего другого сделать нельзя.

Просмотреть список всех доступных модулей, содержащихся в JDK или JRE, можно, запустив команду `java --list-modules`. Чтобы получить детальную информацию о каком-либо модуле, запустите `java --describe-module ${`*`имя_модуля`*`}`. (`${`*`имя_модуля`*`}` — это заместитель с неверным синтаксисом, его необходимо заменить на имя любого модуля по вашему выбору).

Платформенные модули запакованы в JMOD-файлы нового формата, созданного специально для этих целей. Однако код вне JDK также может создавать новые модули. В таком случае говорят о *модульных JAR-файлах*: обычных JAR, которые содержат новую конструкцию, *дескриптор модуля*, определяющий имя модуля, его зависимости и экспорты. Существуют еще модули, создаваемые системой модулей динамически, из JAR, которые не были трансформированы в модули.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

Это приводит к фундаментальному аспекту системы модулей: все на свете — модули! (Или, если быть точным, не имеет значения, какие типы и ресурсы были представлены компилятору или виртуальной машине, поскольку все они заканчиваются модулем.) Модули — это сердце системы модулей и, следовательно, данной книги. Все остальное может быть приведено к ним самим, их именам, определению зависимостей или API, который они экспортируют.

1.4.2. Ваш первый модуль

JDK может быть модуляризован — это прекрасно, но что насчет кода? Как он приводится к модулям? На самом деле все довольно просто.

Единственное, что необходимо добавить, — файл под названием `module-info.java` *(декларацию модуля)* в корневую папку и заполнить его именем данного модуля, его зависимостями от других модулей и пакетами, которые составляют его публичный API:

```
module my.xml.app {
    requires java.base;
    requires java.xml;
    exports my.xml.api;
}
```

Вы увидите, что без требуемого java.base в итоге можно обойтись

Выглядит так, будто модуль *my.xml.app* использует платформенные модули *java.base* и *java.xml* и экспортирует пакет `my.xaml.ap`. Пока все идет хорошо. Сейчас вы скомпилируете `module-info.java` вместе с другим исходным кодом из файлов с расширением `.class` и запакуете это в JAR-файл. (Компилятор и инструмент `jar` выполнят данные операции автоматически.) И вуаля — готов ваш первый модуль.

1.4.3. Система модулей в действии

Запустим XML-приложение и посмотрим на систему модулей в действии. Для этого запустите следующие команды:

```
java
    --module-path mods
    --module my.xml.app
```

Отсюда начинается система модулей. Потребуется несколько шагов, чтобы улучшить ситуацию с комком грязи, которая знакома вам еще из разделов 1.2 и 1.3.

1. Самогенерация.
2. Проверка подключения всех необходимых модулей.
3. Построение внутреннего отображения архитектуры приложения.
4. Вызов метода `main` из начального модуля.
5. Поддержание активности модуля на протяжении всей работы приложения, чтобы защитить внутренние компоненты модуля.

На рис. 1.12 представлены все шаги. Однако не будем забегать вперед и разберем каждый шаг по порядку.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

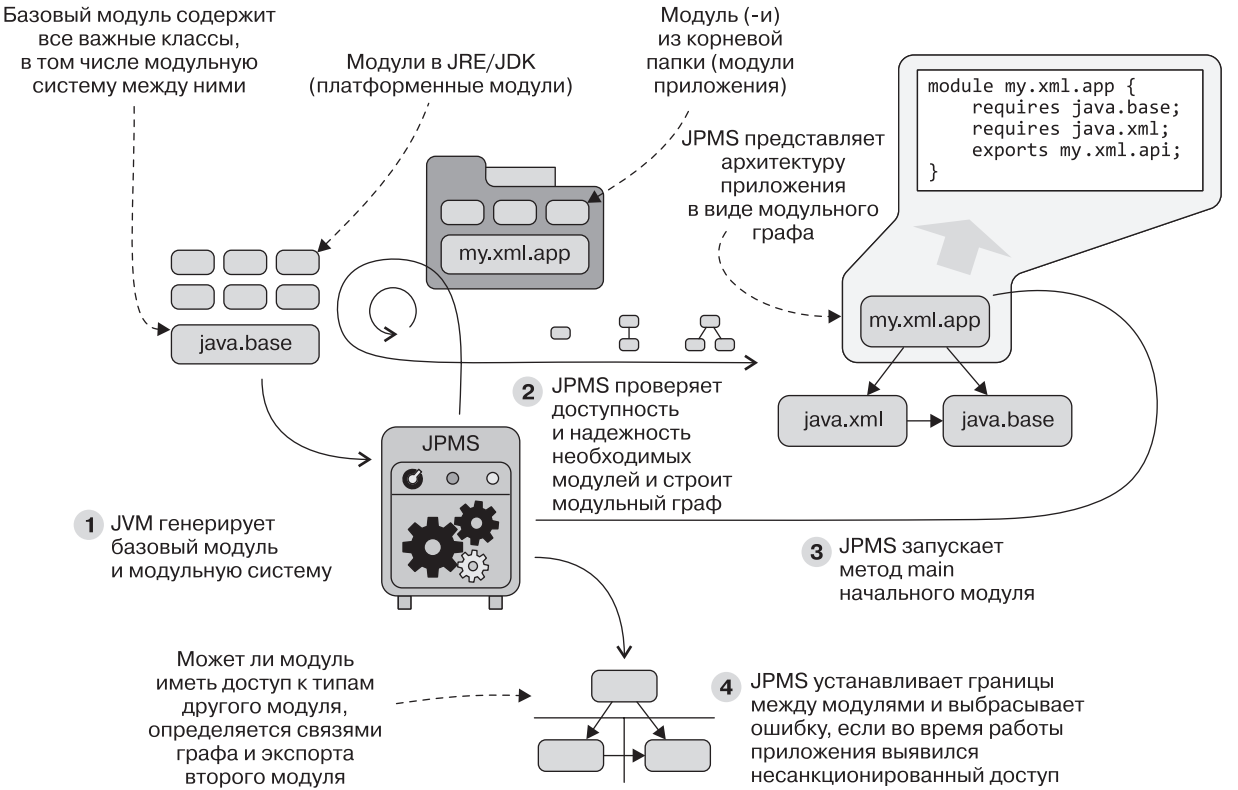


Рис. 1.12. Система модулей платформы Java (JPMS) в действии. Почти все процессы проходят во время запуска: после *(1)* самогенерации она *(2)* убеждается, что все модули подключены, строит модульный граф и *(3)* устанавливает контроль над работающим приложением. Во время выполнения она *(4)* проверяет, что внутренние компоненты каждого модуля находятся под защитой

Загрузка базового модуля

Система модулей — это всего лишь код, и, как вы уже изучили, — модули. Так что же именно содержит JPMS? Это *java.base*, базовый модуль. В лучших традициях вопроса о курице и яйце система модулей и базовый модуль загружают друг друга.

Базовый модуль также представляет собой первый узел графа, построенного JPMS. Это именно то, что он делает дальше.

Модульное решение: построение графа, отображающего приложение

Команда, которую вы запустили, заканчивалась на `--module my.xml.app`. Это дает системе модулей понять, что *my.xml.app* — главный модуль приложения и распределение его зависимостей должно начаться здесь. Но где JPMS может найти модуль? Вот здесь и пригодится `--module-path mods`. Он сообщает системе модулей, что она может найти модули приложения в папке `mods`, так что JPMS прилежно поищет в ней *my.xml.app*.

Однако папки содержат не модули, а JAR-файлы. Поэтому система модулей просканирует все JAR в `mods` и просмотрит все дескрипторы модулей. В примере `mods` содержит *my.xml.app.jar*, и его дескриптор говорит о том, что тот включает модуль под названием *my.xml.app*. Как раз его и ищет система модулей! JPMS создает внутреннее отображение *my.xml.app* и добавляет его в граф модуля — пока еще не содержащий ничего больше.

Система модулей нашла начальный модуль. Что дальше? Поиск его зависимостей. Дескриптор *my.xml.app* определяет, что нужны *java.base* и *java.xml*. Где JPMS может найти их?

Первый, *java.base*, уже известен, поэтому система модулей может добавить связь от *my.xml.app* к *java.base* — первое ребро графа. Далее *java.xml*. Он начинается с *java*, и это указывает системе модулей на то, что перед ней платформенный модуль; как следствие, JPMS не ищет данный модуль по его пути, а вместо этого ищет собственное хранилище модулей. JPMS находит там *java.xml* и добавляет его в граф, связав с ним *my.xml.app*.

Сейчас у вас есть уже три узла графа, но только два из них решены. Зависимости *java.xml* все еще неизвестны, поэтому JPMS проверяет их в следующую очередь. Он не имеет никаких зависимостей, помимо *java.base*, так что распределение модулей можно считать завершенным. Начиная с *my.xml.app* и вездесущего базового модуля, процесс построил небольшой граф с тремя узлами.

Если JPMS не может найти требуемый модуль или сталкивается с какой-либо неопределенностью (например, два JAR-файла содержат модули с одинаковыми именами), то процесс завершается с информативным сообщением об ошибке. Это значит, что вы можете исследовать проблемы во время запуска, которые в иной обстановке могли бы привести к поломке приложения во время работы в какой-то произвольной точке.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Запуск начального модуля

Итак, повторим: как начался процесс? Ах да, с команды, заканчивающейся `--module my.xml.app`. Система модулей выполнила одну из своих ключевых функций — проверила наличие всех необходимых зависимостей — и теперь может взять под контроль все приложение.

Начальный модуль *my.xml.app* — не просто стартовая точка приложения. Он также обязан содержать метод `public static void main(String[])`. Однако вам не обязательно указывать класс, содержащий данный метод, при запуске приложения. Я не упомянул это раньше, но вы прилежно упаковали файлы `.class` в JAR и уже тогда указали основной класс. Информация об этом встроена в дескриптор модуля, и теперь JPMS может ее прочитать.

Поскольку вы использовали `--module my.xml.app` без указания главного класса, система модулей ожидает, что данная информация окажется в дескрипторе модуля. К счастью, так и есть, и она запускает `main` из данного класса. Приложение запускается, но работа JPMS на этом не заканчивается!

Защита содержимого модуля

Даже если приложение успешно запущено, система модулей должна оставаться активной, чтобы поддерживать вторую необходимую функцию: защиту внутренних компонентов модуля. Помните строку `exports my.xml.api` в декларации модуля *my.xml.app*? Это как раз то место, где появляется указанная функция.

Когда модуль впервые получает тип другого модуля, JPMS проверяет выполнение следующих требований:

- ❑ модификатор доступа данного типа должен быть `public`;
- ❑ модуль, которому принадлежит данный тип, должен экспортировать пакет, куда этот тип входит;
- ❑ в графе модуля требующий доступа модуль должен быть присоединен к своему владельцу.

Когда *my.xml.app* впервые использует `javax.xml.XMLConstants` (к примеру), система модулей убеждается, что тип доступа к `XMLConstants` — public (✔), *java.xml* экспортирует `javax.xml` (✔) и *my.xml.app* соединен с *java.xml* в графе модуля (✔). Поскольку все три условия выполняются, *my.xml.app* может использовать `XMLConstants`.

Такое поведение устраняет критический недостаток подхода «комок грязи», применяемого Java для отношений между артефактами: нет никакого способа отделить внутренний для артефакта код от кода, который будет использоваться публично. Когда используется `exports`, модуль может явно определить, какие части API публичные, а какие — внутренние, и может зависеть от системы модулей, которая и примет решение.

Более сложный пример

В качестве менее банального примера возьмем рис. 1.13, где показан граф модуля для приложения *ServiceMonitor*, приведенного на рис. 1.2. Его четыре JAR-файла (*monitor*, *observer*, *statistics* и *persistence*) — как и две зависимости (*spark* и *hibernate*) — были превращены в модули. Модули JDK, такие как *java.xml* и *java.base*, тоже видимы, поскольку приложение зависит и от них.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

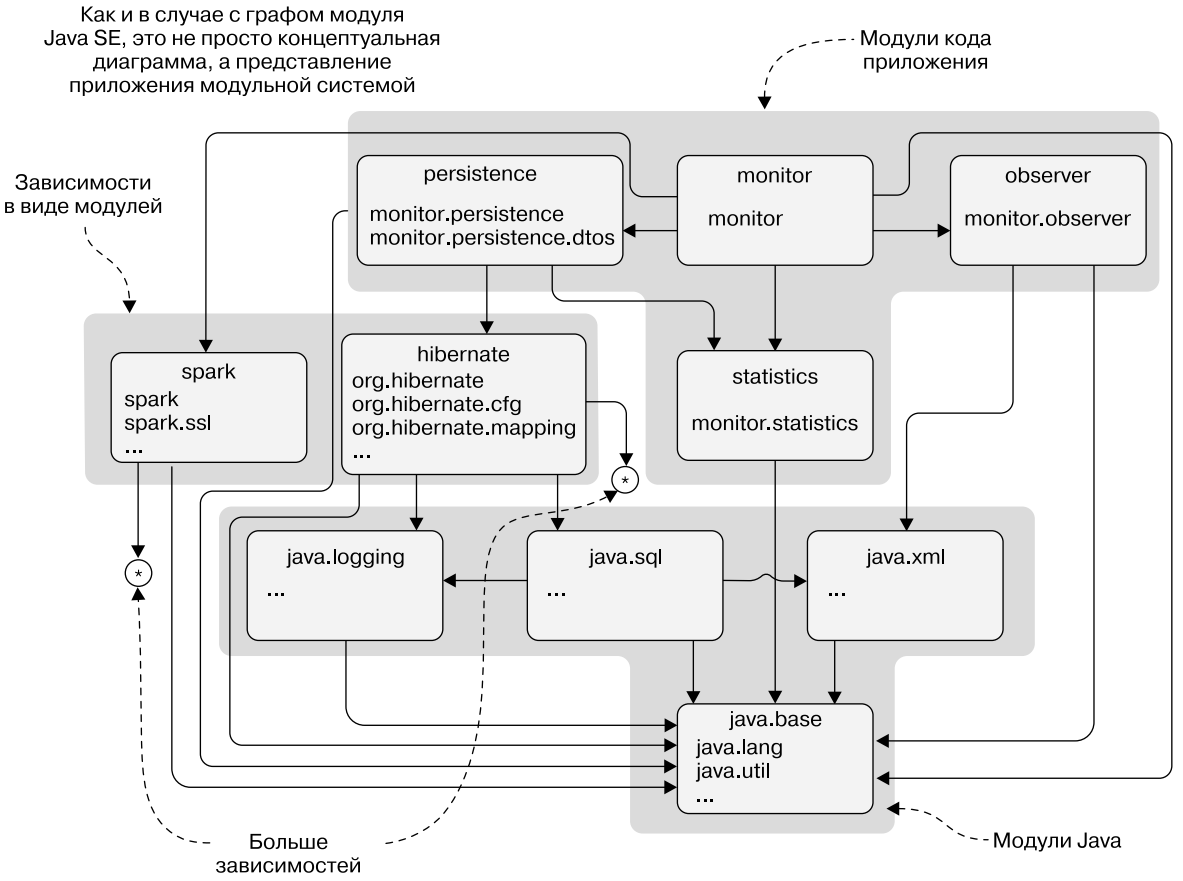


Рис. 1.13. Граф модуля для приложения ServiceMonitor очень похож на архитектурную схему, показанную на рис. 1.6. Граф показывает четыре модуля, содержащих код приложения, две библиотеки, которые он использует, чтобы реализовать нужный функционал, и включенные модули из JDK. Стрелки определяют зависимости между ними. В каждом модуле перечислено несколько пакетов из экспортируемых им

Я нахожу сходство с рис. 1.6, который изображает зависимости между JAR-файлами *ServiceMonitor*. Этот рисунок показывает *наше* понимание того, как приложение организовано на уровне артефактов, в то время как рис. 1.13 отображает видение *системы модулей*. Их удивительное сходство демонстрирует, насколько хорошо система модулей подходит для выражения архитектуры приложения.

1.4.4. Ваш немодульный проект в целом будет в порядке

Разработчики существующих проектов с большими базами кода заинтересуются путями миграции. Хотя системы модулей всегда означают «внутри или снаружи» (то есть для их использования все должно быть представлено в виде модулей), данный способ не подходит для JPMS. Чтобы поддерживать обратную совместимость, обычное приложение, запущенное из пути класса на Java 8, должно делать то же самое на Java 9. Таким образом, немодульные приложения следует запускать поверх модульной среды разработки, что подразумевает способность системы модулей обработать этот случай.

И она это делает. Я уже упоминал мимоходом, что система обрабатывает JAR-файлы, которые еще не были превращены в модули. Это возможно как раз благодаря обратной совместимости. Переход на систему модулей полезен, однако *необязателен*.

Как следствие, путь класса, нужный для определения JAR или простых файлов с расширением `.class` для компилятора и JVM, работает в Java 8 и более ранних версиях. Даже сами модули в пути класса работают так же, как немодульные JAR-файлы. Основное предположение заключается в том, что путь класса отвечает за доступ к артефактам, не желающим превращаться в комок грязи, который мы обсуждали в разделе 1.3.

Параллельно этому была создана новая концепция: *путь модуля*. Здесь же основное предположение состоит в том, что она рассматривает все артефакты как модули. Что интересно, такое видение справедливо даже для обычных JAR-файлов.

ВАЖНАЯ ИНФОРМАЦИЯ

Сосуществование пути класса и пути модуля и соответствующая обработка ими модульных артефактов — ключ к пониманию больших приложений с точки зрения системы модулей. Глава 8 подробно исследует эту важную тему.

Другой аспект системы модулей, особенно важный для устаревших проектов, — это совместимость. JPMS влечет за собой множество закадровых изменений, и хотя большинство из них обратно совместимые в строгом смысле слова, некоторые все же плохо взаимодействуют с существующей кодовой базой. Примеры:

- зависимости от внутренних API JDK (например, из пакетов `sun.*`) становятся причиной ошибок во время компиляции и выполнения;
- API JEE должны быть настроены вручную;
- разные артефакты, классы которых лежат в одном и том же пакете, могут вызвать проблемы;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- компактные профили, механизмы расширения и стандартного переопределения и другие похожие возможности были убраны;
- соответственно был изменен макет образа среды выполнения;
- `URLClassLoader` больше не является загрузчиком класса приложения.

В конце концов, независимо от того, модуляризовано приложение или нет, запуск в Java 9 или более поздней версии может нарушить его работу. Главы 6 и 7 рассказывают об идентификации и решении чаще всего возникающих проблем.

На данном этапе у вас могут появиться такие вопросы.

- Могут ли Maven, Gradle и другие инструменты управлять зависимостями?
- Что насчет инициативы доступа к открытым сервисам (OSGi)? Почему я не могу просто использовать ее?
- Разве система модулей не избыточна во времена, когда все пишут микросервисы?

И вы правы, спрашивая это. Ни одна технология не является обособленным островом, и стоит взглянуть на экосистему Java в целом, изучив, как существующие инструменты и подходы связаны с системой модулей и могут повлиять на нее в будущем. Я уже сделал это в разделе 15.3, вы уже знаете все, что нужно для понимания ответов на вопросы, — так почему бы, если не можете оставить эти вопросы без внимания, не прочитать его сейчас?

Раздел 1.5 описывает высокоуровневые цели, которых хочет достичь система модулей, а глава 2 показывает пример того, как может выглядеть модульное приложение. Главы 3, 4 и 5 детально изучают, как написать, скомпилировать, упаковать и запустить подобные приложения с нуля. Часть II книги обсуждает совместимость и миграцию до того, как часть III доберется до расширенных функций системы модулей.

1.5. Цели системы модулей

Изначально система модулей платформы Java была разработана для того, чтобы научить Java использовать графы зависимостей между артефактами. Идея состоит в том, что если Java прекратит уничтожать модульную структуру, то исчезнут и ужасные последствия такого уничтожения.

В первую очередь это должно уменьшить количество проблемных мест, которые появляются с текущим положением дел. Более того, система модулей предоставляет новые возможности, до сих пор недоступные разработчикам, не использовавшим другие системы модулей. Следовательно, это должно повлиять и на модуляризацию программного обеспечения в целом. Что это значит, если говорить более конкретно?

Пока мы не добрались до самой сути, важно заметить, что не все цели системы модулей одинаково важны для всех типов проектов. Чаще всего выигрывают большие, долгоиграющие проекты, такие как JDK, для которых в целом и была разработана JPMS. Большинство целей вряд ли окажут огромное влияние на ежедневное написание кода, в отличие, например, от лямбда-выражений в Java 8 или `var` в Java 10. Однако они изменят способ написания и развертывания проекта — того, чем мы занимаемся каждый день (не так ли?).

Среди всех целей системы модулей можно выделить две самые важные: надежные настройка и инкапсуляция. Мы рассмотрим их более подробно, чем остальные.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

1.5.1. Надежная настройка: не оставлять после себя JAR

Как видно из подраздела 1.3.3 при рассмотрении системы модулей в действии, отдельные модули объявляют свои зависимости от других модулей и JPMS анализирует эти зависимости. Хотя мы рассматривали только запуск виртуальной машины, один и тот же механизм работает во время и компиляции, и соединения (да, это ново; см. главу 14). Такие операции могут провалиться, если зависимости конфликтуют или вообще отсутствуют. Тот факт, что отсутствие зависимости обнаружится во время запуска, а не в момент запроса ее одним из классов, — уже большая победа.

До Java 9 JAR-файлы с одними и теми же классами не рассматривались как конфликтующие. Вместо этого процесс выполнения выбирал произвольный класс, затеняя другие, что приводило к сложностям, описанным в подразделе 1.3.2. Начиная с Java 9, компилятор и виртуальная машина стали распознавать эти и другие неопределенности, что позволяет выявлять проблему на ранних сроках.

> **Определение: надежная настройка**
>
> В совокупности это делает *настройку системы* более надежной, чем раньше, поскольку лишь очень хорошо сформированные настройки запуска пройдут данный тест. Если они сделают это, то виртуальная машина сможет превратить концептуальный граф зависимостей в граф модуля, который заменит комок грязи на структурированное изображение работающей системы — так же, как сделали бы мы сами.

1.5.2. Надежная инкапсуляция: делаем внутренний код модуля недоступным

Следующая ключевая цель системы модулей — *позволить модулям надежно инкапсулировать внутренний код* и экспортировать лишь определенный функционал.

> Класс, являющийся приватным в модуле, должен быть таким же приватным, как поле, которое приватно в классе. Другими словами, границы модуля должны определять не только видимость классов и интерфейсов, но и их доступность.
>
> *Марк Рейнхольд, Project Jigsaw: Bringing the Big Picture into Focus (mreinhold.org/blog/jigsaw-focus)*

Чтобы достичь этой цели, и компилятор, и виртуальная машина обеспечивают строгие правила доступа в пределах модуля: возможен только доступ к публичным членам (имеются в виду классы и методы) публичных типов в *экспортированных пакетах*. Другие типы недоступны извне модуля — даже с помощью рефлексии. В заключение мы можем строго инкапсулировать внутренние компоненты библиотеки — и быть уверенными в том, что приложения не будут случайным образом зависеть от деталей их реализации.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В целях предотвращения случайных привязок к типам в непрямых зависимостях, которые могут измениться при запуске, ситуация усложняется даже больше: в общем, модуль может получить доступ к тем типам модулей, от которых ему требуется зависеть. (Отдельные расширенные функции создают намеренные исключения из этого правила.)

1.5.3. Автоматическая безопасность и улучшенное удобство сопровождения

Надежная инкапсуляция внутренних API в модуле может значительно улучшить безопасность и удобство сопровождения. Это помогает в вопросах безопасности, поскольку важный код становится эффективно скрытым от другого кода, которому не нужно его использовать. Это также делает сопровождение более простым, ведь теперь гораздо проще не допускать чрезмерного разрастания публичных API.

> Обычное использование API, которые являются внутренними для реализации Java SE Platform, — это одновременно и риск безопасности, и нагрузка при сопровождении. Надежная инкапсуляция, обеспеченная предложенной спецификацией, позволит компонентам, реализующим Java SE Platform, предотвратить доступ к их внутренним API.
>
> *Запрос спецификации Java (JSR) 376*

1.5.4. Улучшенная производительность при запуске

Более четко определенные границы кода позволяют более эффективно использовать существующие методы оптимизации.

> Многие опережающие время методы оптимизации всей программы могут быть более эффективными, если известно, что класс может обращаться только к классам из нескольких других определенных компонентов, а не ко всем классам во время выполнения.
>
> *JSR 376*

Кроме того, возможно индексировать классы и интерфейсы по их аннотациям так, чтобы эти классы можно было найти без полного сканирования их путей. Это еще не было реализовано в Java 9, но вполне может появиться в будущих выпусках.

1.5.5. Масштабируемая платформа Java

Прекрасным следствием появления модулей с четко определенными зависимостями можно назвать то, что теперь легко распознать работающие подмножества JDK. Серверные приложения, к примеру, не используют AWT, Swing или JavaFX и потому могут работать на JDK без этого функционала. Новый инструмент `jlink`

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

(см. главу 14) делает возможным создание приложений только с необходимыми для работы модулями. Можно даже добавить модули библиотек и приложения, тем самым создав самостоятельную программу, которая не требует Java, установленного в хост-системе.

Определение: масштабируемая платформа

Благодаря модульности JDK можно *отобрать* нужный функционал и создать среду выполнения, состоящую только из необходимых модулей.

Это делает Java центральной фигурой при выборе платформы для небольших устройств и контейнеров.

1.5.6. Не цели

К сожалению, система модулей не является панацеей и не подходит для многих интересных случаев. Для начала JPMS не поддерживает концепции версионности. Нельзя создать версию модуля или указать необходимую версию для зависимостей. Конечно, можно встроить подобную информацию в дескриптор модуля и затем получать ее, используя API рефлексии, но это всего лишь метаданные для разработчиков и инструментов — система модулей никак их не обработает.

Поскольку JPMS «не видит» версии, это также говорит о том, что она не различит две версии одного и того же модуля. С другой стороны, и в соответствии с целью надежной настройки она будет воспринимать данную ситуацию как классическую двусмысленность — один и тот же модуль, представленный дважды, — и откажется его компилировать и запускать. Чтобы узнать больше о версиях, обратитесь к главе 13.

JPMS не предоставляет никаких механизмов для поиска и загрузки действующих модулей из централизованного репозитория или для публикации новых модулей. Эта задача в достаточной мере покрывается существующими инструментами сборки.

Целью модулей не является и создание динамических графов, где артефакты возникают или исчезают по мере работы программы. Тем не менее такую систему можно реализовать с помощью уже существующей расширенной функции — слоев (см. раздел 12.4).

1.6. Навыки, старые и новые

Я дал много обещаний, а оставшаяся часть книги объяснит, как система модулей платформы Java стремится их выполнить. Однако не заблуждайтесь, эти преимущества не появляются просто так! Чтобы создавать приложения поверх системы модулей, вам придется поразмышлять о ее артефактах и зависимостях гораздо больше, чем раньше, и посвятить множество мыслей коду. Некоторые функции, работавшие раньше, перестанут работать в Java 9, и использование отдельных структур потребует чуть-чуть больше усилий, нежели раньше.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Можно сравнить это с тем, насколько сильно и статически типизируемый язык требует увеличения объема работы, чем динамически типизируемый, — как минимум во время написания кода. Все эти типы и обобщения — разве нельзя просто взять `Object` и использовать его повсюду? Конечно, можно, но готовы ли вы отказаться от безопасности, которую предоставляет система типов, ради того, чтобы просто приложить чуть меньше усилий при написании кода? Мне так не кажется.

1.6.1. Чему вы научитесь

Потребуются новые навыки! К счастью, эта книга им и обучит. Когда все будет сказано и сделано и вы освоите механизмы, изложенные в следующих главах, ни одно из существующих приложений не посмеет бросить вам вызов.

Часть I, особенно главы 3–5, описывает базовые принципы системы модулей. Вдобавок к практическим навыкам вы освоите основные механизмы, которые помогут достичь более глубокого понимания системы модулей. После этого вы сможете описывать модули и отношения между ними, инкапсулируя внутренние компоненты модулей и выражая их зависимости. С помощью `javac`, `jar` и `java` вы скомпилируете, упакуете и запустите модули и приложения, построенные из них.

Часть II книги строится на базовых концепциях, расширяя их, чтобы охватить более сложные случаи использования. Для существующих приложений вы сможете проанализировать вероятные несовместимости с Java 9–11 и запланировать переход на систему модулей, применяя различные функции, которые она предлагает для этого. С такой целью, а также чтобы построить более сложные отношения между модулями, вы сможете задействовать расширенные функции, например квалифицированный экспорт, открытые модули и сервисы, равно как и расширенный API рефлексии. Благодаря применению `jlink` вы создадите урезанные JRE, оптимизированные для конкретных случаев использования, или автономные образы приложений, которые поставляются с собственными JRE. В конце концов вы увидите полную картину, включая то, как система модулей взаимодействует с загрузкой классов, рефлексией и контейнерами.

1.6.2. Что нужно знать

Когда дело доходит до требований к навыкам, JPMS проявляет интересные особенности. Большинство из них совершенно новы и имеют собственный синтаксис, выделенный в объявлении модуля. Изучить его не сложно при наличии базовых навыков Java. Так, если вы знаете, что код организуется в типы, пакеты и в конечном счете в JAR, как модификаторы доступа, в частности `public`, работают между ними, что делают `javac`, `jar` и `java`, а также имеете общее представление о способах использования всего этого — у вас достаточно знаний для понимания части I, равно как и большинства расширенных функций из части III.

Однако для того, чтобы по-настоящему понять проблемы, от которых система модулей собирается избавиться, и оценить предлагаемые решения, требуется нечто большее. Ознакомление со следующими концепциями и опыт работы с большими

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

приложениями упростит понимание возможностей системы модулей и их преимущества и недостатки:

- принципы работы JVM, в частности загрузчика классов;
- проблемы, которые вызывает этот механизм (вспомните JAR-ад);
- более продвинутые API Java, такие как загрузчик сервисов и API рефлексии;
- инструменты сборки, такие как Maven и Gradle, и построение с их помощью проекта;
- способы модуляризации программных систем.

Однако вне зависимости от вашего опыта вы можете встретить отсылки или объяснения, которые никак не связаны с уже имеющимися у вас сведениями. Для такой огромной экосистемы, как Java, это нормально, и каждый может узнать нечто новое, куда бы ни направился (поверьте, я знаю это на собственном примере). Так что никогда не отчаивайтесь! Если корявые объяснения не помогают, то, возможно, вы поймете некоторые технические аспекты, только взглянув на пример кода.

Поскольку все перспективы обрисованы, пришло время запачкать руки и изучить основы JPMS. Я рекомендую продолжить обучение с главы 2, которая представляет собой краткую версию части I и показывает код, определяющий, проектирующий и запускающий модульные JAR-файлы. Она также вводит демоприложение, которое мы будем использовать на протяжении всей книги. Если же вы предпочитаете сначала изучить теорию, то перейдите к главе 3, объясняющей фундаментальные механизмы системы модулей. Если вами движет опасение насчет совместимости вашего проекта с Java 9, то главы 6 и 7 разрешат все возможные проблемы, но в них будет сложно разобраться, не имея хорошего понимания основ.

Резюме

- Систему программного обеспечения можно представить в виде графа, который часто показывает (не)желаемые свойства системы.
- Java не поддерживает графы на уровне JAR-файлов. Это приводит к различным проблемам, среди которых можно выделить JAR-ад, ручную проверку безопасности и неудобства сопровождения.
- Система модулей платформы Java существует для того, чтобы обеспечить понимание графов Java, что вносит в язык модульность на уровне артефактов. Самые важные ее цели — надежные настройка и инкапсуляция, так же как улучшение безопасности, удобства сопровождения и производительности.
- Все это достижимо благодаря вводу модулей — грубо говоря, JAR-файлов с дополнительным дескриптором. Компилятор и среда выполнения интерпретируют описанную информацию, чтобы построить граф зависимостей и предоставить обещанные преимущества.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

2 Структура модульного приложения

В этой главе

- Выкладка исходного кода модульного приложения.
- Создание декларации модуля.
- Компиляция модулей.
- Запуск модульного приложения.

Эта глава познакомит вас с полным циклом создания модульных приложений, но мы не станем детально углубляться в тему. Подробности можно будет найти в главах 3, 4 и 5. Однако с такой всеобъемлющей темой, как система модулей, легко за деревьями не увидеть леса. Именно поэтому сначала стоит рассмотреть цельную картину. Это даст вам понимание того, как разные фрагменты пазла соединяются, представляя отдельное модульное приложение, как описываются и компилируются его модули и оно запускается.

Это значит, что я сразу брошу вас в омут с головой: не все из описанного ниже покажется очевидным. Но если что-то не ясно, то не волнуйтесь — всему вскоре найдется объяснение. Когда вы осилите часть I книги, весь код в примерах обретет смысл. Поэтому загибайте уголки страниц, поскольку, вполне вероятно, вы захотите вернуться к некоторым разделам.

Раздел 2.1 объясняет, что делает гипотетическое приложение, из каких типов оно состоит и что входит в его сферу ответственности. Система модулей появляется в разделе 2.2, где обсуждается, как организовать файлы и папки, описать модули, скомпилировать и запустить приложение. Это краткое знакомство

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

продемонстрирует многие из основных механизмов системы модулей, а также ряд случаев, когда базовых функций не хватает для модуляризации сложного приложения, — тема, которая обсуждается в разделе 2.3. Вы можете найти приложение онлайн по ссылкам www.manning.com/books/the-java-module-system и github.com/CodeFX-org/demo-jpms-monitor. Ветка master содержит вариант, описанный в разделе 2.2.

2.1. Знакомство с ServiceMonitor

Чтобы увидеть систему модулей в действии, нужен пример проекта, к которому ее можно применить. Не так важно, что делает данное приложение, поэтому не беспокойтесь о деталях.

Представим сеть сервисов, которые объединяются, чтобы радовать пользователей, — скажем, социальная сеть или видеоплатформа. Вы хотите понаблюдать за этими сервисами, чтобы определить, насколько безопасна система, и выявить проблемы в момент их появления (а не тогда, когда о них сообщат пользователи). Вот здесь и пригодится пример приложения.

Демоприложение называется *ServiceMonitor*. Оно контактирует с отдельными сервисами, собирает и объединяет данные для диагностики и делает их доступными через REST.

ПРИМЕЧАНИЕ

> Вы можете вспомнить приложение из раздела 1.2 или рис. 1.16, где оно разделено на четыре отдельных JAR-файла. Со временем мы перейдем к еще более детальной информации, но это уже тема для раздела 2.2. Пока подумаем, как реализовать такую систему в виде цельного артефакта (назовем это монолитным подходом). Неважно, если сказанное пойдет вразрез с главой 1, — новая глава, новые детали.

К счастью, сервисы уже самостоятельно собирают необходимые нам данные, так что *ServiceMonitor* нужно будет лишь периодически их запрашивать. Это работа для экземпляров класса `ServiceObserver`. Как только информация для диагностики появится в форме `DiagnosticDataPoint`, ее можно будет передать в `Statistician`, который соберет все в `Statistics`. Статистика, в свою очередь, хранится в `StatisticsRepository` и доступна с помощью REST. А класс `Monitor` связывает все это вместе.

Рисунок 2.1 показывает отношение всех этих типов друг к другу. Чтобы лучше понять, как это работает, посмотрим на пример кода, начиная с интерфейса `ServiceObserver` (листинг 2.1).

Листинг 2.1. Интерфейс ServiceObserver

```
public interface ServiceObserver {

  DiagnosticDataPoint gatherDataFromService();

}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

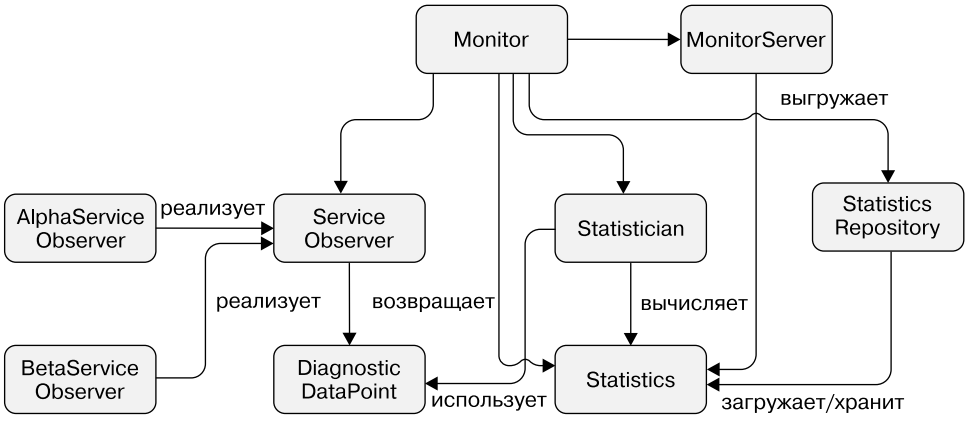


Рис. 2.1. Классы, из которых состоит приложение ServiceMonitor. Два экземпляра ServiceObserver запрашивают сервисы через Alpha и Beta API и возвращают данные для диагностики, которые Statistician собирает в Statistics. Статистика хранится и загружается в репозитории, а также выгружается через REST API. Monitor дирижирует всем этим

Выглядит достаточно просто, но, к сожалению, не все сервисы можно выгрузить через один и тот же REST API. Используются два поколения: Alpha и Beta. Вот почему `ServiceObserver` — это интерфейс с двумя реализациями (рис. 2.2): каждый экземпляр соединяется с разными поколениями API и следит за тем, чтобы все данные выгружались в ваше приложение через соответствующий интерфейс.

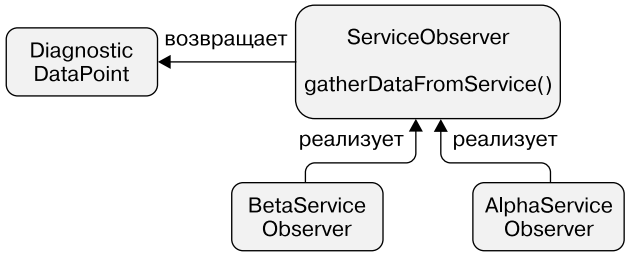


Рис. 2.2. Рассматриваемый сервис имеет два поколения API для выгрузки диагностических данных. Соответственно ServiceObserver интерфейс имеет две реализации

Класс `Statistician` не хранит собственное состояние — он всего лишь предоставляет два метода, которые создают новый экземпляр `Statistics` или комбинируют существующую статистику и новые данные в обновленную статистику (листинг 2.2).

Листинг 2.2. Класс Statistician

```
public class Statistician {

 public Statistics emptyStatistics() {
  return Statistics.empty();
 }
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
 public Statistics compute(
   Statistics currentStats,
   Iterable<DiagnosticDataPoint> dataPoints) {
  Statistics finalStats = currentStats;
  for (DiagnosticDataPoint dataPoint : dataPoints)
    finalStats = finalStats.merge(dataPoint);
  return finalStats;
 }

}
```

Класс `StatisticsRepository` не делает ничего особенного — загружает и хранит статистику (листинг 2.3). Как он это делает — с помощью сериализации, JSON-файлов или резервной базы данных, — для нашего приложения не имеет значения.

Листинг 2.3. Класс StatisticsRepository

```
public class StatisticsRepository {

 public Optional<Statistics> load() { /* ... */ }

 public void store(Statistics statistics) { /* ... */ }

}
```

Вы получаете тип, который собирает данные, конвертирует их в статистику и затем хранит ее. Что мы пропустили, так это класс, который соединяет их между собой, регулярно подтягивая данные и загружая их через `Statistician` в репозиторий. Для этого и был создан класс `Monitor`. Листинг 2.4 показывает его поля и метод `updateStatistics()`, исполняющий его основное предназначение. (Код, который проверяет регулярный запуск задачи, опущен.)

Листинг 2.4. Класс Monitor и его метод updateStatistics()

```
public class Monitor {

 private final List<ServiceObserver> serviceObservers;
 private final Statistician statistician;
 private final StatisticsRepository repository;
 private Statistics currentStatistics;
 // [...]

 private void updateStatistics() {
  List<DiagnosticDataPoint> newData = serviceObservers
    .stream()
    .map(ServiceObserver::gatherDataFromService)
    .collect(toList());
  Statistics newStatistics = statistician
    .compute(currentStatistics, newData);
  currentStatistics = newStatistics;
  repository.store(newStatistics);
 }

 // [...]
}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Класс `Monitor` хранит текущую статистику в поле `currentStatistics` (типа `Statistics`).

Во время запроса класс `MonitorServer`, который выгружает REST API, просит монитор предоставить статистические данные — либо из памяти, либо из сохраненного состояния — и затем выгружает запрошенные биты и возвращает их (листинг 2.5).

Листинг 2.5. Класс MonitorServer

```
public class MonitorServer {

 private final Supplier<Statistics> statistics;

 public MonitorServer(Supplier<Statistics> statistics) {
  this.statistics = statistics;
 }

 // [...]

 private Statistics getStatistics() {
  return statistics.get();
 }

// [...]

}
```

Интересна такая деталь: класс `MonitorServer` вызывает `Monitor`, но не зависит от него. Это происходит потому, что `MonitorServer` получает ссылку не на монитор, а на поставщик информации, который передает вызов монитору. Причина проста: `Monitor` управляет целым приложением, что делает его классом, внутри которого происходит множество событий. Я не хотел соединять REST API с таким тяжеловесным объектом только для вызова отдельного геттера. До Java 8 я бы, наверное, создал специальный интерфейс, чтобы получить статистику и реализовать ее с помощью класса `Monitor`. Однако с появлением Java 8 лямбда-выражения и существующие функциональные интерфейсы значительно упростили необходимую развязку.

В конечном итоге мы определили следующие типы:

- `DiagnosticDataPoint` — данные о доступности сервиса за определенный период времени;
- `ServiceObserver` — интерфейс для обзора сервисов, возвращающий `DiagnosticDataPoint`;
- `AlphaServiceObserver` и `BetaServiceObserver` — оба рассматривают варианты сервисов;
- `Statistician` — вычисляет статистику из `DiagnosticDataPoint`;
- `Statistics` — содержит полученную статистику;
- `StatisticsRepository` — хранит и предоставляет статистику;
- `MonitorServer` — отвечает на REST-запросы для статистики;
- `Monitor` — связывает все вместе.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

2.2. Модуляризация ServiceMonitor

Если бы вы реализовали приложение *ServiceMonitor* так же, как оно было описано, в реальном проекте, то внедрение системы модулей в полную силу было бы равноценно попытке расколоть орех кувалдой. Но это всего лишь пример, призванный показать анатомию данной системы, вследствие чего мы модуляризуем его так, как если бы это был гораздо больший проект.

Раз речь зашла о структуре, начнем с разделения приложения на модули, прежде чем обсудить, как исходный код размещается в файловой системе. Затем перейдем к самому интересному: объявлению, компиляции и запуску приложения.

2.3. Разделение ServiceMonitor на модули

Самый распространенный способ модуляризации приложения — это разделение его по задачам. *ServiceMonitor* выполняет следующие задачи с указанными в скобках связанными типами:

- сбор данных из сервисов (`ServiceObserver`, `DiagnosticDataPoint`);
- объединение данных в статистику (`Statistician`, `Statistics`);
- сохранение статистики (`StatisticsRepository`);
- выгрузка статистики через REST API (`MonitorServer`).

Однако не только доменная логика определяет требования. Кроме вышеперечисленных, есть еще технические задачи:

- сбор данных должен быть скрыт за API;
- Alpha- и Beta-сервисы требуют отдельных реализаций данного API (`AlphaServiceObserver` и `BetaServiceObserver`);
- все задачи должны быть управляемыми (`Monitor`).

ВАЖНАЯ ИНФОРМАЦИЯ

Все это выливается в следующие модули с указанными публично доступными типами:

- monitor.observer (ServiceObserver, DiagnosticDataPoint);
- monitor.observer.alpha (AlphaServiceObserver);
- monitor.observer.beta (BetaServiceObserver);
- monitor.statistics (Statistician, Statistics);
- monitor.persistence (StatisticsRepository);
- monitor.rest (MonitorServer) * monitor (Monitor).

Наложив эти модули на диаграмму классов, показанную на рис. 2.3, можно легко увидеть появление зависимостей между модулями.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

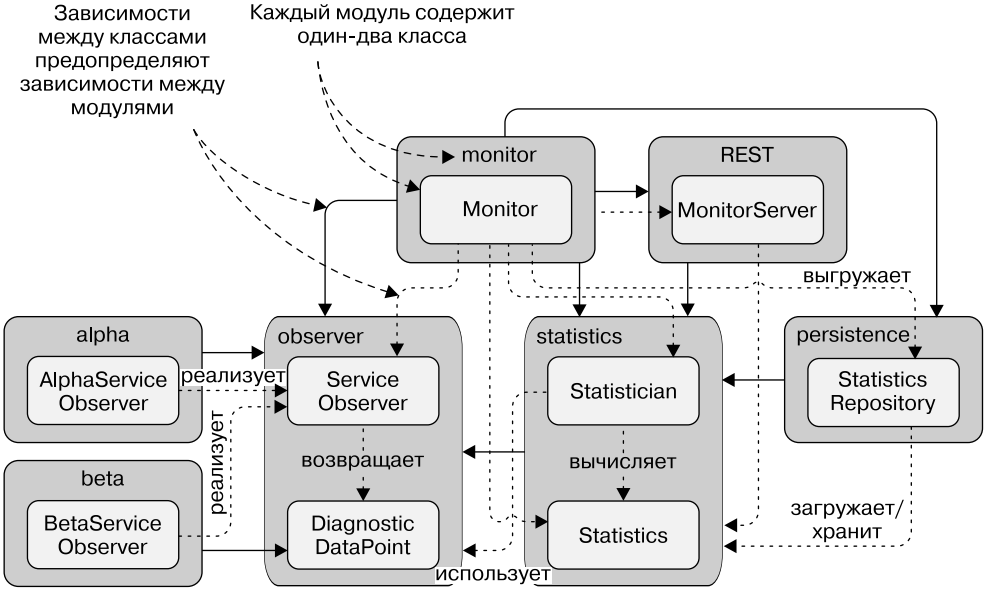


Рис. 2.3. Модули приложения ServiceMonitor (обычные стрелки) накладываются на структуру классов (пунктирные стрелки). Обратите внимание на то, как межклассовые зависимости за границами модулей определяют модульные зависимости

2.4. Организация файлов в структуре каталогов

На рис. 2.4 показана структура каталогов приложения. Каждый модуль представляет собой отдельный проект. Это значит, он будет иметь отдельную структуру каталогов. Нет никаких причин усложнять что-либо, поэтому можно использовать одну и ту же структуру для всего. Если вы уже участвовали в различных проектах или использовали Maven, Gradle или другие инструменты сборки, то вам все это покажется знакомым.

В первую очередь стоит обратить внимание на папку `mods`. Позже, при создании новых модулей, в ней они и будут храниться. Раздел 4.1 более подробно расскажет о структуре каталогов.

Далее следует несколько необычная папка `libs`, которая содержит сторонние зависимости. В реальных проектах она вам не понадобится, поскольку инструменты сборки сами будут управлять зависимостями. Однако вы собираетесь проделать это вручную, так что проще всего будет собрать все зависимости в одном месте. Таким образом, это не рекомендация и даже не требование — всего лишь упрощение.

Следующее необычное явление — `module-info.java`. Это так называемая *декларация модуля*, и она отвечает за описание свойств модуля. Она располагается

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

прямо в центре системы модулей и в самом сердце данной книги, особенно раздела 3.1. Тем не менее мы пока только бросим на нее беглый взгляд в следующем разделе.

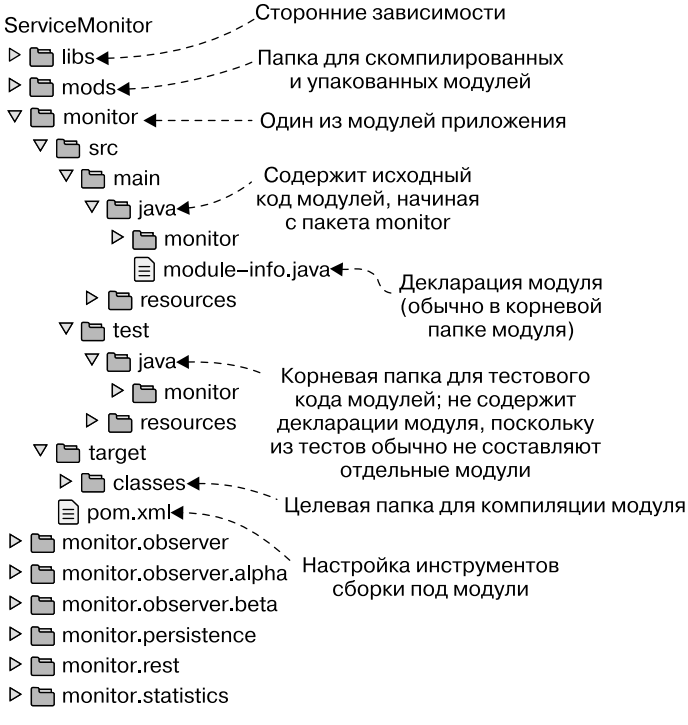


Рис. 2.4. Каждый модуль приложения ServiceMonitor — это отдельный проект со своей легкоузнаваемой структурой каталогов. Новое — папка mods, в которой содержатся модульные JAR-файлы после сборки и файл декларации модуля module-info.java в корневом каталоге каждого проекта

2.5. Объявление и описание модулей

ВАЖНАЯ ИНФОРМАЦИЯ

У каждого модуля есть декларация. По соглашению это файл module-info.java в корневой папке проекта. Из него компилятор создает дескриптор модуля, module-info.class. Когда скомпилированный код пакуется в JAR-файл, дескриптор должен находиться в корневой папке, чтобы система модулей могла найти и обработать его.

Как уже обсуждалось в разделе 2.3, приложение состоит из семи модулей, поэтому должно получиться семь деклараций модулей. Листинг 2.6 отображает их все. Даже не зная никаких деталей, можно мельком взглянуть и понять, что происходит.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Листинг 2.6. Декларации для всех модулей ServiceMonitor

```
module monitor.observer {
 exports monitor.observer;
}

module monitor.observer.alpha {
 requires monitor.observer;
 exports monitor.observer.alpha;
}

module monitor.observer.beta {
 requires monitor.observer;
 exports monitor.observer.beta;
}

module monitor.statistics {
 requires monitor.observer;
 exports monitor.statistics;
}

module monitor.persistence {
 requires monitor.statistics;
 requires hibernate.jpa;
 exports monitor.persistence;
 exports monitor.persistence.entity;
}

module monitor.rest {
 requires spark.core;
 requires monitor.statistics;
 exports monitor.rest;
}

module monitor {
 requires monitor.observer;
 requires monitor.observer.alpha;
 requires monitor.observer.beta;
 requires monitor.statistics;
 requires monitor.persistence;
 requires monitor.rest;
}
```

Блок `module the.name { }` объявляет модуль. Его имя обычно следует соглашению о именовании пакетов: оно должно быть уникальным в глобальном смысле, содержать доменное имя, быть написанным в нижнем регистре, а разделы должны быть отмечены точками (подробнее — в подразделе 3.1.3, я использовал укороченные имена только затем, чтобы сделать их более подходящими для книги). Внутри модульного блока директивы `requires` выражают зависимости между модулями, а директивы `exports` объявляют публичный API каждого модуля, перечисляя пакеты, чьи публичные типы могут быть экспортированы.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

2.5.1. Объявление зависимостей от других модулей

ВАЖНАЯ ИНФОРМАЦИЯ

Директива requires содержит имя модуля и сообщает JVM, что объявляемый модуль зависит от модуля, указанного в директиве.

Видно, что реализация обозревателя зависит от API обозревателя, что, конечно, имеет смысл. Модуль *statistics* также зависит от API обозревателя, поскольку `Statistician::compute` использует тип `DiagnosticDataPoint`, который является частью API.

Подобно этому, модуль *persistence* нуждается в статистике, так что зависит от модуля *statistics*. Кроме того, он зависит от Hibernate, поскольку использует его для обращения к базе данных.

Затем мы переходим к *monitor.rest*, который тоже зависит от модуля *statistics*, поскольку тот обрабатывает статистику. Вдобавок он использует микрофреймворк Spark для создания конечной точки REST. Во время модуляризации приложения в разделе 2.1 я отметил, что класс `MonitorServer` не зависит от класса `Monitor`. Это сейчас пригодится, поскольку означает, что *monitor.rest* не зависит от *monitor*. И это хорошо, так как *monitor* зависит от *monitor.rest*, а система модулей запрещает объявление циклических зависимостей. В итоге *monitor* зависит от всех остальных модулей, ввиду того что создает множество экземпляров и передает результаты от одного к другому.

2.5.2. Объявление публичных API модуля

ВАЖНАЯ ИНФОРМАЦИЯ

Директива exports содержит имя пакета и информирует JVM о том, что другие модули, зависимые от объявляемого, могут видеть публичные типы из этого пакета.

Большинство модулей экспортируют единственный пакет: тот, который содержит типы, перечисленные во время определения модуля. Вы могли заметить, что имена пакетов всегда начинаются с имен модулей — часто они идентичны. Так делать не обязательно, но, поскольку имена модулей и пакетов следуют обратной схеме доменных имен, это распространенный случай.

Модуль состояния — единственный, что экспортирует больше одного пакета. В дополнение к *monitor.persistence*, который содержит его основные особенности (`StatisticsRepository`), он экспортирует и *monitor.persistence.entity*. Данный пакет определяет набор классов, аннотированных таким образом, чтобы Hibernate понимал, как их загружать и хранить (обычно их называют сущностями). То есть у Hibernate должен быть доступ к ним; это, в свою очередь, подразумевает, что модуль должен экспортировать пакет, содержащий их. (Если вы полагаетесь на рефлексию Hibernate приватных полей или конструкторов, то экспорта будет недостаточно — см. раздел 12.2 для поиска решения.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Еще одно исключение — *monitor*, который не экспортирует ни одного пакета. Это имеет смысл, поскольку он сидит, подобно пауку в сети, в центре модульного графа и управляет потоком выполнения. Поэтому он и не имеет API, к которому обращались бы другие модули. Наличие главного модуля — который обычно и естественно содержит главный метод программы и не экспортирует никаких пакетов — распространенная практика.

2.5.3. Визуализация ServiceMonitor с помощью модульного графа

Аккуратно объявив зависимости и экспорт модулей, посмотрим на получившийся в итоге модульный граф, показанный на рис. 2.5. Он выглядит всего лишь как подправленная версия рис. 2.3, однако является чем-то гораздо большим! Рисунок 2.3 — это схема, которую архитекторы приложения нарисовали бы на доске. Он показывает модули и отношения между ними, но это только плоды вашего воображения — ничего, что имело бы значение для компилятора или виртуальной машины. Рисунок 2.5, с другой стороны, — это интерпретация системой модулей вашей архитектуры.

То, что модульные аспекты на обеих схемах так похожи и даже в каком-то смысле взаимозаменяемы, означает возможность выразить свое видение архитектуры приложения весьма точно в коде: а именно, в декларациях модулей. Разве это не прекрасно?

Написание кода осталось таким же, как раньше

Вам наверняка интересно, изменился ли процесс написания кода после Java 9. Ответ — нет в абсолютном большинстве случаев. Однако изменились некоторые детали, и главы 6 и 7 подробно их освещают, чтобы вы были подготовлены.

Помимо непосредственной модуляризации проекта и периодических размышлений о том, в какой пакет поместить классы и стоит ли изменять зависимость или экспорт, ежедневная работа по созданию домена и решению проблем останется неизменной. С поддержкой IDE изменить зависимости или экспорты стало так же просто, как управлять импортами пакетов.

Однако полная картина того, как организовать большие кодовые базы, значительно упростилась. Добавление зависимостей, особенно тех из них, которые уже находились в переходном состоянии, стало более явным и потому более легко обсуждаемым во время парного программирования, просмотра кода или архитектуры — это гарантирует, что предполагаемая и реальная архитектура не смогут развалиться просто так. Характерные возможности, например сервисы (см. главу 10) и модули-агрегаторы (см. подраздел 11.1.5), расширят набор инструментов модульности, что приведет к улучшенному проектированию при правильном использовании.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

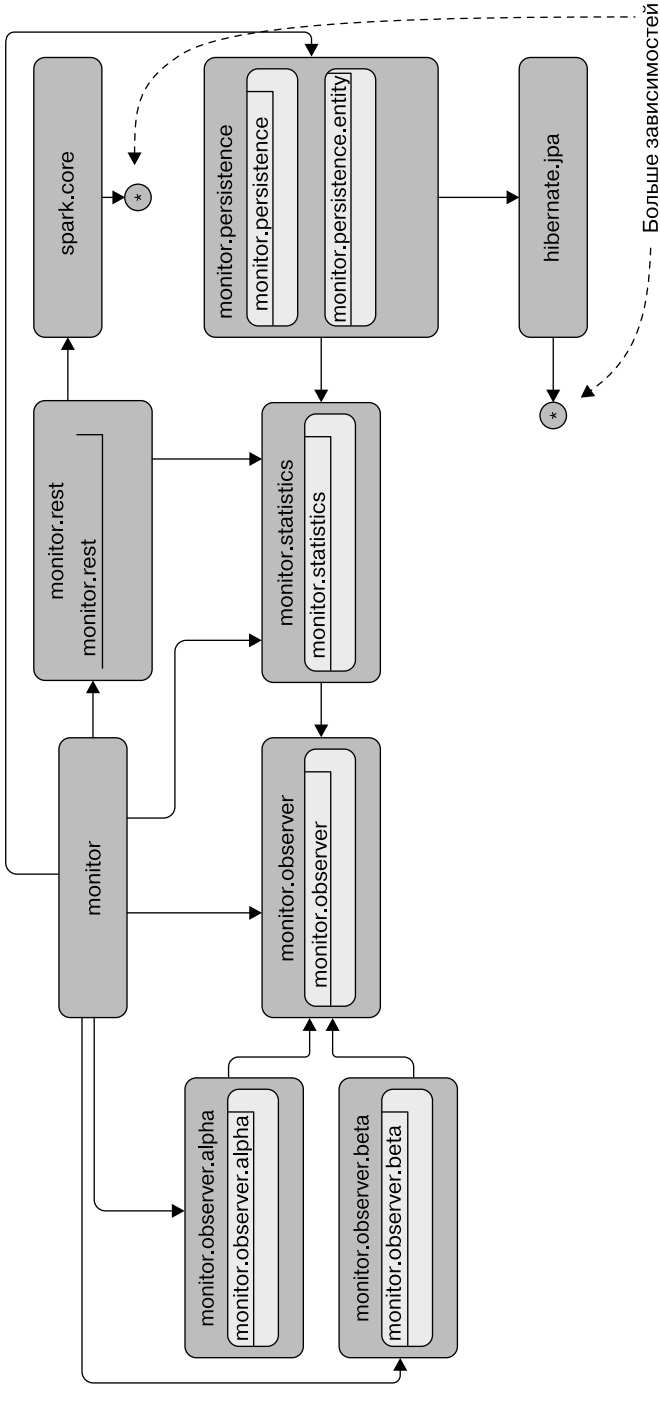


Рис. 2.5. Модульный граф приложения, показывающий модули с экспортируемыми ими пакетами и зависимостями между ними. В отличие от рис. 2.3, это не просто архитектурная схема; здесь показано, как система модулей видит приложение

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

2.6. Компиляция и упаковка модулей

К этому времени вы уже аккуратно организовали проект в специфичные для модулей папки, создали декларации модулей и написали код, который уже можно использовать для сборки и (позже) запуска приложения. Чтобы собрать его, вы создадите модульные артефакты; это процесс из двух шагов: компиляции и упаковки.

Во время компиляции компилятору понадобится знать, где располагаются модули, на которые ссылаются их декларации. Это обычное дело для зависимостей от модулей Java, поскольку компилятор знает, где их найти (в файлах библиотек/модулей среды выполнения).

ВАЖНАЯ ИНФОРМАЦИЯ

Чтобы ваши собственные модули можно было найти, используйте путь модуля — концепцию, параллельную пути класса, но, как указывает ее имя, ожидающую модульные JAR-файлы вместо обычных JAR. Он будет просканирован, когда компилятор начнет искать указанные модули. Объявлять путь модуля в javac можно с помощью новой опции: --module-path, или коротко -p. (Аналогичный подход используется для запуска приложения с JVM. Соответственно те же опции, --module-path и -p, были добавлены в Java, где они выполняют одну и ту же функцию.)

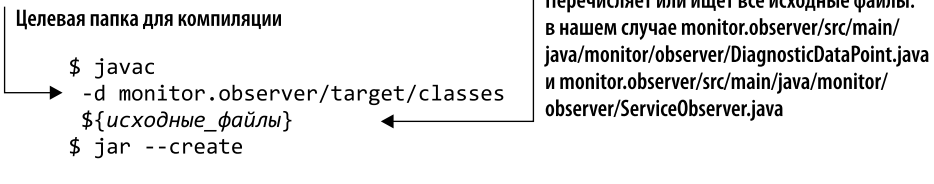
Для хранения модулей мы использовали папку `mods`, что означает две вещи:

- путь модуля будет содержать `mods`;
- упакованные артефакты будут созданы в `mods`.

Некоторые из ваших модулей имеют внешние зависимости: модуль состояния использует Hibernate (`hibernate.jpa`), а модулю REST необходим Spark (`spark.core`). На данный момент проще всего предположить, что эти артефакты также являются уже модуляризованными JAR-файлами и вы или инструмент разместили их в папке `mods`.

Что произойдет, если поместить обычный JAR в путь модуля, модульный JAR в путь класса или смешать их? И как можно поступить, если вам требуется зависимость, которая еще не была модуляризована? Все это — отдельные части истории миграции, и они объясняются в главе 8.

Со всеми предварительными условиями в `mods` можно уже скомпилировать и упаковать приложение. Начать стоит с *monitor.observer*, у которого нет зависимостей. Он не содержит ничего нового — выполнение этих же команд в Java 8 приведет к точно такому же результату:



Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
        --file mods/monitor.observer.jar
        -C monitor.observer/target/classes .
```

Дает название всем новым файлам в mods

Компилирует исходные файлы

У модуля *monitor.alpha* есть зависимость, вследствие чего нужно использовать путь модуля, чтобы сообщить компилятору, где можно найти требуемые артефакты. Естественно, на упаковку с помощью `jar` это не повлияет:

```
$ javac --module-path mods
 -d monitor.observer.alpha/target/classes
 ${исходные_файлы}
$ jar --create
 --file mods/monitor.observer.alpha.jar
 -C monitor.observer.alpha/target/classes .
```

Папка, в которой javac будет искать модули, от которых зависит код

Большинство оставшихся модулей работают абсолютно аналогично. Одно исключение — *monitor.rest*, у которого есть сторонние зависимости в папке `libs`, так что это нужно добавить в путь модуля:

```
$ javac --module-path mods:libs
 -d monitor.rest/target/classes
 ${исходные_файлы}
```

Модуль имеет зависимости в двух папках, поэтому обе они были добавлены в путь модуля

Еще одно исключение — это *monitor*. У вас есть возможность проинформировать систему модулей о том, что он содержит метод `main`, который может использоваться как точка входа в приложение:

```
$ javac --module-path mods
 -d monitor/target/classes
 ${исходные_файлы}
$ jar --create
 --file mods/monitor.jar
 --main-class monitor.Monitor
 -C monitor/target/classes .
```

Класс, в котором содержится метод main приложения

Рисунок 2.6 показывает результат всего этого. Новые JAR-файлы выглядят так же, как и старые обычные JAR, с одним исключением: каждый содержит дескриптор модуля `module-info.class`, который указывает на то, что это модульный JAR.

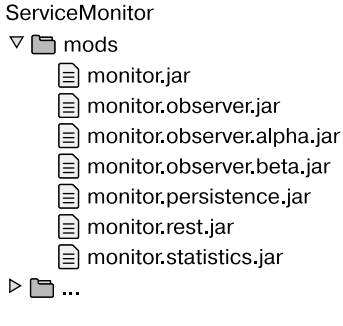


Рис. 2.6. Все модули приложения, скомпилированные и упакованные в mods, готовы к запуску

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

2.7. Запуск ServiceMonitor

Когда все модули скомпилированы в папку `mods`, приложение наконец готово к запуску. Как видно из следующей строки, это то самое место, где окупается тяжелая работа, которую вы вложили в объявление модуля:

```
$ java
 --module-path mods:libs
 --module monitor
```

Папка, в которой java ищет модули

Название модуля для запуска

ВАЖНАЯ ИНФОРМАЦИЯ

Все, что осталось сделать, — это вызвать команду java, указать путь модуля, чтобы java знала, где найти артефакты, из которых состоит ваше приложение, и объявить ей, какой модуль нужно запустить. Разрешение всех зависимостей, проверка отсутствия всех конфликтов или неоднозначных ситуаций и запуск только с правильным набором модулей обрабатываются системой модулей.

2.8. Расширение модульной кодовой базы

Конечно, ни один программный проект не может считаться завершенным (пока не умрет), поэтому изменения неизбежны. Что произойдет, если, например, вы захотите добавить еще одну реализацию обозревателя? Обычно потребуются следующие шаги.

1. Разработайте подпроект.
2. Убедитесь, что он собран.
3. Используйте его в существующем коде.

И это как раз то, что вы сделаете прямо сейчас. Система модулей обретет смысл для нового модуля, если вы добавите его декларацию:

```
module monitor.observer.gamma {
 requires monitor.observer;
 exports monitor.observer.gamma;
}
```

Скомпилируйте и упакуйте новый модуль точно так же, как остальные:

```
$ javac --module-path mods
 -d monitor.observer.gamma/target/classes
 ${исходные_файлы}
$ jar --create
 --file mods/monitor.observer.gamma.jar
 -C monitor.observer.gamma/target/classes .
```

Затем можно будет добавить его к остальным в виде зависимости:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
module monitor {
 requires monitor.observer;
 requires monitor.observer.alpha;
 requires monitor.observer.beta;
 requires monitor.observer.gamma;
 requires monitor.statistics;
 requires monitor.persistence;
 requires monitor.rest;
}
```

На этом все. Учитывая, что сборка берет на себя компиляцию и упаковку, вам нужно всего лишь добавить или изменить декларацию модуля. Это верно и для удаления или рефакторинга модулей: помимо обычной работы, которая потребуется, необходимо будет потратить еще немного времени на обдумывание того, как это повлияет на модульный граф, и на соответствующее изменение деклараций.

2.9. Разбор полетов: эффекты системы модулей

В целом все прошло хорошо, не правда ли? Прежде чем углубляться в детали в следующих главах, уделим немного времени рассмотрению двух преимуществ, обещанных системой модулей, а также ряда нюансов, которые вы наверняка надеетесь сгладить с помощью более сложных функций.

2.9.1. Что система модулей делает для вас

Обсуждая цели системы модулей в разделе 1.5, мы говорили о двух наиболее важных: надежных настройке и инкапсуляции. Сейчас, когда вы уже собрали нечто более или менее конкретное, мы можем пересмотреть эти цели и понять, как они помогут выпустить надежное, удобное в сопровождении программное обеспечение.

Надежная настройка

Что произойдет, если зависимость пропадет из `mods`? Если две зависимости потребуют две разные версии одного и того же проекта — например, Log4j или Guava? Что случится, если два модуля экспортируют одни и те же типы случайно или намеренно?

В пору пути класса эти проблемы проявляли себя во время выполнения приложения: одни прекращали его работу, в то время как другие были более коварными и неуловимыми, что, возможно, привело бы к поломке поведения приложения. Когда в игру вступила система модулей, многие из этих ненадежных ситуаций, в частности упомянутые мною, стали распознаваться гораздо раньше. Компилятор или JVM прекратят работу, выдав детализированное сообщение, что даст вам шанс исправить ошибку.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Вот, например, сообщение, которое будет получено, если приложение запустится, но *monitor.statistics* утеряется:

```
> Error occurred during initialization of boot layer
> java.lang.module.FindException:
>     Module monitor.statistics not found,
>     required by monitor
```

Аналогично вот результат запуска *ServiceMonitor* с двумя SLF4J-версиями, представленными в одном пути модуля:

```
> Error occurred during initialization of boot layer
> java.lang.module.FindException:
>     Two versions of module org.slf4j.api found in mods
>     (org.slf4j.api-1.7.25.jar and org.slf4j.api-1.7.7.jar)
```

Теперь вы тоже можете перестать случайно зависеть от зависимостей ваших зависимостей. Hibernate использует SLF4J; это значит, что библиотека всегда доступна на время работы приложения. Но, как только вы начинаете импортировать типы из SLF4J (незатребованные ни в какой из деклараций модуля), компилятор остановит вас, информируя о том, что вы пытаетесь использовать код из модуля, от которого явно не зависите:

```
> monitor.persistence/src/main/java/.../StatisticsRepository.java:4:
>     error: package org.slf4j is not visible
>     (package org.slf4j is declared in module org.slf4j.api,
>      but module monitor.persistence does not read it)
```

Даже если обмануть компилятор, система модулей запустит те же проверки во время запуска.

Надежная инкапсуляция

Теперь посмотрим на ситуацию со стороны не пользователя, а сопровождающего. Представьте рефакторинг кода в *monitor.observer.alpha*, возможно, как подготовку к исправлению ошибки или улучшение производительности. После выпуска новой версии вы обнаруживаете, что некий код из *monitor* сломан, вследствие чего приложение неустойчиво. Если вы меняли публичный API, то это ваша вина.

Но что, если вы изменили внутреннюю реализацию в типе, который был доступен, несмотря на пометку «неподдерживаемый»? Возможно, этот тип должен был быть публичным, поскольку вы планировали использовать его в двух пакетах, или, к примеру, авторы *monitor* получали доступ к нему через рефлексию?

С помощью системы модулей данной ситуации можно избежать. Вы уже сделали это! Только экспортируемые типы в пакетах являются видимыми. Все остальное надежно защищено — даже от рефлексии.

ПРИМЕЧАНИЕ

Если вам интересно, что нужно сделать для действительного перехода на систему модулей, то обратитесь к разделу 7.1 и подразделу 12.2.2.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

2.9.2. Что еще модульная система может сделать для вас

Несмотря на довольно неплохую модуляризацию *ServiceMonitor*, осталось еще несколько недостатков, которые нужно обсудить. Сейчас с ними ничего нельзя сделать, но расширенные функции, представленные в части III книги, позволят сгладить их. Этот раздел познакомит вас с тем, что еще только предстоит изучить.

Маркировка незаменимых модульных зависимостей

Модули *monitor.observer.alpha* и *monitor.observer.beta* объявляют зависимости от *monitor.observer*, что имеет смысл, поскольку они реализуют объявленный в нем `ServiceObserver`. Кроме того, они возвращают экземпляр `DiagnosticDataPoint`, который также ему принадлежит.

Это приводит к интересным последствиям для любого кода, который использует модули реализации:

```
ServiceObserver observer = new AlphaServiceObserver("some://service/url");
DiagnosticDataPoint data = observer.gatherDataFromService();
```

Модуль, в котором содержатся эти строки, должен также зависеть от *monitor.observer*, иначе не сможет увидеть типы `ServiceObserver` и `DiagnosticDataPoint`. Даже сам модуль *monitor.observer.alpha* становится практически бесполезным для клиентов, не зависящих от *monitor.observer*.

Наличие модуля, который можно использовать только в том случае, если его клиенты не забыли явно провести зависимость от другого модуля, — не слишком удобно. Хорошо, что есть способ это обойти! Вы узнаете о подразумеваемой читабельности в разделе 11.1.

Разделение реализаций и клиентов API

Размышление о взаимоотношениях между *monitor.observer* и модулями реализаций, *monitor.observer.alpha* и *monitor.observer.beta* наводит на новую мысль. Почему *monitor* должен знать о реализациях?

В его нынешнем виде ему нужно создать экземпляры конкретных классов, но с тех пор он взаимодействует с ними только через интерфейс. Зависимость от целого модуля ради одного конструктора выглядит избыточной. Более того, каждый раз, когда устаревшая реализация `ServiceObserver` свернется или будет введена новая реализация, потребуется обновить зависимости модуля *monitor* и затем перекомпилировать, упаковать и развернуть артефакт.

Чтобы обеспечить более слабую связь между реализациями и клиентами API, где таким клиентам, как *monitor*, не придется зависеть от реализаций наподобие *monitor.observer.alpha* и *monitor.observer.beta*, система модулей представляет сервисы. Их обсуждение начинается с главы 10.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Повышение целенаправленности экспорта

Помните, как модуль состояния экспортировал пакет, который содержал объект для передачи данных, затем использовавшийся в Hibernate?

```
module monitor.persistence {
 requires monitor.statistics;
 requires hibernate.jpa;
 exports monitor.persistence;
 exports monitor.persistence.entity;
}
```

Код выглядит не совсем правильным — только Hibernate нужно видеть эти сущности. Однако сейчас и другие модули, зависящие от *monitor.persistence*, например *monitor*, могут их видеть.

И опять все это можно покрыть расширенными функциями системы модулей. Квалифицированные экспорты позволят модулю экспортировать пакеты не для всех, а для ограниченного списка модулей. Раздел 11.3 представляет этот механизм.

Создание пакетов, доступных только для рефлексии

Даже экспортирование пакета только для определенных модулей может быть избыточным.

- Вы можете скомпилировать модуль с помощью API (такого как Java Persistence API, JPA), а не конкретной реализации (например, Hibernate), поэтому по понятным причинам не стоит упоминать модуль реализации в квалифицированном экспорте.
- Вы можете использовать инструмент на основе рефлексии (вроде Hibernate или Guice), который имеет доступ к коду во время выполнения через рефлексию, так зачем делать его доступным во время компиляции?
- Возможно, вам нужна рефлексия приватных членов (Hibernate делает это во время настройки для внедрения полей), что не будет работать в экспортированных пакетах.

Решение представлено в разделе 12.2, который вводит *открытые модули* и *открытые пакеты*. Они делают пакеты доступными только во время выполнения и в обмен на это позволяют использовать рефлексию приватных членов, что часто требуется инструментам на основе рефлексии. Подобно экспорту, они *квалифицированно открыты*, с помощью чего вы можете открыть пакет лишь для выбранных модулей.

Если вы пользовались, к примеру, Hibernate в роли JPA-поставщика, то наверняка вам пришлось усердно поработать, чтобы предотвратить прямые зависимости от Hibernate. В этом случае прямое записывание зависимости в декларацию модуля явно не то, что нужно. Подраздел 12.3.5 подробно обсуждает данный сценарий.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

2.9.3. Добавление необязательных зависимостей

Нередки случаи, когда проект содержит код, который выполняется только в случае представления определенной зависимости в запущенном приложении. Модуль *monitor.statistics*, к примеру, может содержать код, задействующий причудливую библиотеку статистики, которая — возможно, из-за проблем с лицензией — не всегда присутствует во время запуска *ServiceMonitor*. Другой пример — библиотека с опциями, интересными пользователю только в случае применения сторонней зависимости — например, тестируется фреймворк, который объединяется с библиотекой утверждений (assertion library), работающей только в паре с фреймворком.

В указанных случаях в соответствии со сказанным ранее все зависимости должны быть определены в декларации модуля. Благодаря этому они будут доступны во время компиляции — для успешного завершения процесса. К сожалению, та же директива `requires` означает, что зависимость также должна быть представлена во время запуска, или же JVM откажется запускать приложение.

Это нас не устраивает. Но, как и ожидалось, система модулей презентовала решение: необязательные зависимости, которые должны присутствовать во время компиляции, но не требуются во время выполнения. Они обсуждаются в разделе 11.2. После того как мы обсудим все эти расширенные функции, раздел 15.1 покажет вариант *ServiceMonitor*, в котором будет использоваться большинство из них.

Каждому из трех шагов — объявления, сборки и запуска модульного приложения — посвящена отдельная глава: 3, 4 и 5 соответственно. Все они важны, но глава 3 особенно, поскольку знакомит с фундаментальными концепциями и механизмами, лежащими в основе системы модулей.

Резюме

- Во время модуляризации приложения можно вывести модульные зависимости из зависимостей типов через границы модулей. Это упрощает создание начального графа зависимостей модуля.
- Структура каталогов проекта с несколькими модулями похожа на ту, что использовалась до Java 9, поэтому существующие инструменты и подходы продолжат работать.
- Декларация модуля — файл `module-info.java` в корневом каталоге проекта — самое очевидное изменение, которое система модулей внесла в написание кода. Она именует модули и объявляет как зависимости, так и публичные API. Кроме этого, способ написания кода практически не изменился.
- Команды `javac`, `jar` и `java` были обновлены для работы с модулями. Самое очевидное и часто применяемое изменение — путь модуля (функция `--module-path` или `-p` в командной строке). Он подобен пути класса, но используется для модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

3 Определение модулей и их свойств

В этой главе

- Что такое модули и как их описывает декларация модуля.
- Различия между типами модулей.
- Модульная читабельность и доступность.
- Понимание пути модуля.
- Построение модульного графа с урегулированием модулей.

Мы уже долго обсуждаем модули. Они являются не только строительными блоками для модульных приложений, но и ключом к правильному осмыслению системы модулей. Поэтому важно достичь глубокого понимания того, что это вообще такое и как их свойства влияют на поведение программы.

Из трех основных шагов определения, сборки и запуска модулей данная глава исследует первый (остальные два описываются в главах 4 и 5). Здесь подробно объясняется, что такое модуль и как его декларации описывают его имя, зависимости и API (см. раздел 3.1). Пара примеров из JDK помогут вам составить первое впечатление о модульном ландшафте, который мы собираемся исследовать с Java 9, а также выделят типы модулей для лучшей навигации.

Мы также обсудим, как система модулей — и, как следствие, компилятор и среда выполнения — взаимодействует с модулями (см. разделы 3.2 и 3.3). И наконец, что немаловажно, разберем путь модуля и то, как система модулей разрешает зависимости и строит из них граф (см. раздел 3.4).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Чтобы получить примеры кода, обратитесь к ветке master в *ServiceMonitor*. Она содержит большинство деклараций модуля, показанных в данной главе. В завершение вы узнаете, как объявить имя модуля, его зависимости и API, и то, как система модулей ведет себя на основании этой информации. Вы станете понимать сообщения об ошибках, которые может выдать данная система, сможете проанализировать их и устранить главную проблему.

Указатели

Данная глава закладывает основу для всего, и потому вся оставшаяся часть книги связана с ней. Чтобы сделать эти связи видимыми, глава содержит множество ссылок, ведущих к материалу следующих глав. Если они вам мешают, то проигнорируйте их — они будут важны, когда вы откроете книгу, чтобы найти ту или иную информацию.

3.1. Модули: строительные блоки модульных приложений

После всех этих возвышенных бесед о модулях приходит время запачкать руки. Сначала мы посмотрим на два формата, в которых могут быть представлены модули (JMOD и модульный JAR), прежде чем перейти к объявлению свойств модуля. Закладывая основу для более легкого обсуждения оставшейся части книги, разделим модули на несколько типов.

3.1.1. Модули Java (JMOD), поставляемые с JDK

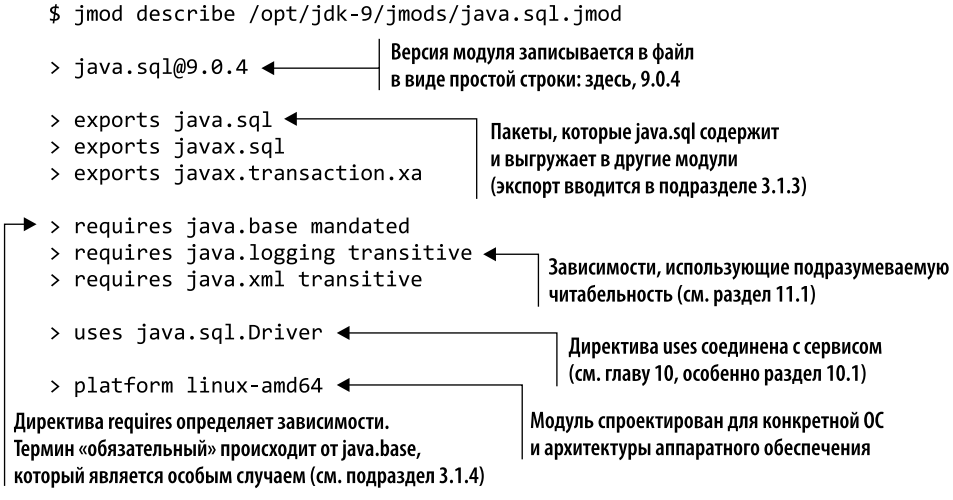
Во время работы над Project Jigsaw кодовая база Java разделилась примерно на 100 модулей, которые были выведены в новый формат под названием *JMOD*. Он нарочно не был специфицирован, чтобы иметь возможность эволюционировать более энергично, чем это было с форматом JAR (который изначально был ZIP-файлом). Он зарезервирован для использования JDK, поэтому мы не станем обсуждать его подробно.

Хотя у нас нет возможности создавать модули JMOD, мы можем их исследовать. Чтобы увидеть модули, содержащиеся в JRE или JDK, вызовите `java --list-modules`. Информация будет получена из оптимизированного модульного хранилища, файла `modules` в папке `libs`, во время установки. JDK (не JRE) также содержат «сырые» модули в папке `jmods`; и новый инструмент `jmod`, который находится в папке `bin`, следующей за `jmods`, можно использовать для вывода свойств с помощью операции `describe`.

Следующий фрагмент показывает пример исследования файла JMOD. Здесь `jmod` используется для описания *java.sql* на Linux-устройстве, где JDK 9 установлен в `/opt/jdk-9`. Подобно большинству модулей Java, *java.sql* применяет несколько

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

расширенных функций системы модулей, поэтому не все подробности могут быть понятны до конца данной главы:



3.1.2. Модульные файлы JAR: доморощенные модули

Мы не можем создавать модули JMOD, как же тогда выгрузить созданный модуль? Для этого и были созданы модульные JAR-файлы.

> **Определение: модульный JAR и дескриптор модуля**
>
> *Модульный JAR* — это обычный JAR, за исключением одной маленькой особенности. Его корневой каталог содержит *дескриптор модуля*: файл `module-info.class`. (В данной книге JAR без модулей называются *обычными JAR*, но это не официальный термин.)

Дескриптор модуля содержит всю информацию, необходимую системе модулей для создания представления модуля во время выполнения. Все свойства отдельного модуля представлены в этом файле, вследствие чего многие особенности, которые обсуждаются на протяжении книги, внесли в него свой вклад. Создание такого дескриптора из исходного файла, как описано в следующем разделе, и включение его в JAR позволяет разработчикам и инструментам создавать модули.

Хотя дескриптор модуля позволяет модульному JAR-файлу быть чем-то большим, чем просто архив файла класса, использование его в таком ключе не обязательно. Клиент может применять его как обычный JAR, игнорируя все модульные свойства и включив его в путь класса. Это просто незаменимо для постепенной модуляризации существующих проектов (раздел 8.2 знакомит вас с *безымянным модулем*).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

3.1.3. Декларация модулей: определение свойств модулей

Итак, для превращения любого старого JAR в модуль вам потребуется только дескриптор модуля, `module-info.class`. Это, однако, ведет к вопросу, как создать дескриптор. Расширение `.class` подразумевает, что дескриптор — результат компиляции исходного файла.

Определение: декларация модуля

Модульный дескриптор компилируется из *декларации модуля*. По соглашению это файл `module-info.java` в корневой папке проекта. Декларация — это центральный элемент модулей и системы модулей в целом.

Декларация против дескриптора

Вам может быть интересно, как не запутаться в терминах *«декларация модуля»* и *«дескриптор модуля»*. Это не такая уж большая проблема. Первое — это исходный код, а последнее — байт-код, но они являются лишь разными формами одной и той же идеи: нечто, описывающее свойства модуля. Контекст обычно подразумевает только один вариант, поэтому обычно понятно, что имелось в виду.

Если это вас не удовлетворило и вы хотите всегда использовать правильный вариант, то я могу помочь, предложив свою мнемонику: лексикографически *декларация* находится до *дескриптора*, что легко запомнить, поскольку сначала идет исходный код, а за ним байт-код. Теперь все в нужном порядке: сначала декларация/исходный код, затем дескриптор/байт-код.

Декларация модуля определяет его идентифицируемость и поведение в системе модулей. Большинство особенностей, описанных в следующих главах, имеют соответствующие директивы в декларации модуля и будут описаны в свое время. Теперь же посмотрим на три основных свойства, отсутствующие в файлах JAR: имя, явные зависимости и инкапсулированные внутренние компоненты.

ВАЖНАЯ ИНФОРМАЦИЯ

Ниже представлена структура простого module-info.java-файла, который объявляет три этих свойства:

```
module ${имя_модуля} {
 requires ${имя_модуля};
 exports ${имя_пакета};
}
```

Естественно, `${имя_модуля}` и `${имя_пакета}` должны быть заменены на настоящие имена модуля и пакета.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Возьмем для примера дескриптор модуля *monitor.statistics* из *ServiceMonitor*:

```
module monitor.statistics {
 requires monitor.observer;
 exports monitor.statistics;
}
```

Легко можно распознать структуру, которую я только что описал: за ключевым словом `module` следует имя модуля, а тело содержит директивы `requires` и `exports`. В следующих разделах будет подробно рассмотрено объявление этих свойств.

Новые ключевые слова?

Вам может быть интересно, что означают новые ключевые слова, такие как `module`, `requires`, `exports` и другие в последующих главах, для кода, который уже использует эти термины в названиях полей, параметров, переменных и других именованных сущностей. К счастью, нам не о чем волноваться. Это ключевые слова *для служебного использования* — они применяются в качестве ключевых слов только в том случае, если синтаксически находятся там, где нужно. Так что, несмотря на то что вы не можете задействовать переменную с именем `package` и модуль с именем `byte`, можно использовать переменную и даже модуль с именем `module`.

Именование модулей

Самое основное свойство, опускаемое в JAR-файлах, — имя, которое компилятор и JVM могут использовать для идентификации. Так что это самая очевидная характеристика модуля. У вас будет возможность и даже обязательство дать имя каждому созданному модулю.

ВАЖНАЯ ИНФОРМАЦИЯ

В дополнение к ключевому слову module декларация начинается с именования модуля. Имя должно быть идентификатором, и потому к нему будут применяться те же правила, что и при именовании пакета. Имена модулей обычно создаются в нижнем регистре и иерархически структурированы точками.

Именование модулей будет вполне естественным, поскольку большинство инструментов, которыми вы пользуетесь ежедневно, уже принуждают вас давать имя каждой составляющей проекта. Однако имеет смысл применять имя проекта в качестве отправной точки в поисках имени модуля. Очень важно выбирать мудро!

Как вы увидите в разделе 3.2, система модулей сильно клонится в сторону имени модуля. Конфликты и особенно эволюция имен ведут к проблемам, поэтому важно, чтобы имя было:

- уникальным в глобальном смысле;
- устойчивым.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Лучший способ достичь этого — использовать обратную схему доменных имен, которая часто применяется для пакетов. С учетом ограничения идентификаторов это часто приводит к тому, что имя модуля используется в качестве префикса для пакетов, которые в него включены. Это не обязательно, но является хорошим доказательством того, что оба имени были выбраны осознанно.

Синхронизация имени модуля и префикса имени пакета подчеркивает: изменение имени модуля (что подразумевало бы изменение имени пакета) — одно из наиболее серьезных возможных изменений. В интересах стабильности это должно случаться исключительно редко.

К примеру, следующий дескриптор именует модуль *monitor.statistics* (ради краткости модули, из которых состоит приложение *ServiceMonitor*, не следуют обратной схеме доменных имен):

```
module monitor.statistics {
 // урезанные требования и экспорты
}
```

Все другие свойства объявляются внутри фигурных скобок за именем модуля. Никакой определенный порядок не требуется, но в целом принято начинать с зависимостей, прежде чем перейти к экспорту.

Требования модулей для выражения зависимостей

Еще одна вещь, которую мы пропустили в JAR, — это возможность объявить зависимости. Во время использования файлов JAR мы никогда не можем знать, какой артефакт им нужен, чтобы правильно работать, и зависим ли мы от инструментов сборки или документации для проверки. В системе модулей зависимости обязаны быть явными. (Рисунок 3.1 помогает понять, как это работает.)

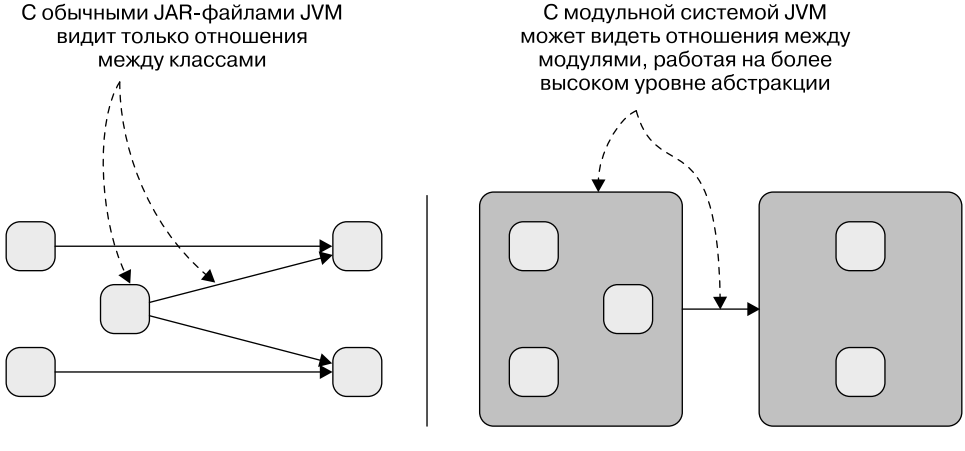


Рис. 3.1. Возможность выразить зависимости между модулями вводит новый уровень абстракции, который JVM может рассмотреть. Без них *(слева)* она видит только зависимости между типами; но с ними *(справа)* она видит зависимости между артефактами так, как видим их мы

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Определение: зависимости

Зависимости обозначаются с помощью директивы `requires`, содержащей ключевое слово, за которым следует имя модуля. Директива утверждает, что объявленный модуль зависит от именованного и требует его во время компиляции и выполнения.

Модуль *monitor.statistics* имеет зависимость во время компиляции и выполнения от *monitor.observer*, которая объявляется с помощью директивы `requires`:

```
module monitor.statistics {
 requires monitor.observer;
 // урезанные экспорты
}
```

В случае объявления зависимости с помощью директивы `requires` система модулей выдаст ошибку, если не найдет модуля с таким именем. Компиляция, как и запуск, прервется при потере модуля (см. раздел 3.2).

Экспорт пакетов для объявления API модуля

И наконец, экспорт, который определяет публичный API модуля. Здесь вы можете выбрать, какие пакеты содержат типы, которые должны быть доступны извне модуля, и какие предназначены только для внутреннего использования.

Определение: экспортированные пакеты

За ключевым словом `exports` следует имя пакета, который включен в модуль. Только *экспортированные пакеты* доступны извне модуля; все остальные надежно инкапсулированы в нем (см. раздел 3.3).

Модуль *monitor.statistics* экспортирует пакет с таким же именем:

```
module monitor.statistics {
 requires monitor.observer;
 exports monitor.statistics;
}
```

Обратите внимание: несмотря на то что мы можем так думать, пакеты не являются иерархическими! Пакет `java.util` не содержит `java.util.concurrent`, соответственно, экспортирование первого не включает в себя экспортирование последнего. Строка импорта, `import java.util.*`, импортирует все типы из `java.util`, но не из `java.util.concurrent` (рис. 3.2).

Это значит, что если модуль захочет экспортировать два пакета, то ему понадобятся две директивы `exports`. Система модулей не предполагает никаких шаблонов наподобие `exports java.util.*`, чтобы это упростить, — выгрузка API должна быть осознанным действием.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

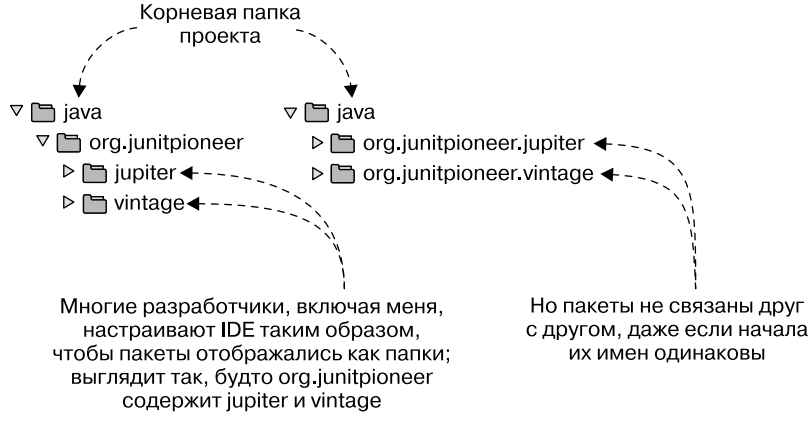


Рис. 3.2. Мы обычно представляем пакеты иерархическими, где org.junitpioneer содержит extension и vintage *(слева)*. Однако не в этом случае! Для Java имеют значение только полные имена пакетов, и он не видит разницы между этими двумя *(справа)*. Это нужно принимать во внимание во время экспортирования пакетов. Например, экспортирование exports org.junitpioneer не будет включать в себя экспортирование каких-либо типов из jupiter или vintage

Примеры деклараций модуля

Чтобы сделать первый шаг, рассмотрим пару настоящих деклараций модуля. Самый фундаментальный модуль — *java.base*, поскольку он содержит `java.lang.Object` — класс, без которого ни одна Java-программа не будет работать. На этой зависимости заканчиваются все остальные: она необходима всем модулям, но сама не требует ни одного. Зависимость от *java.base* настолько фундаментальна, что модули даже не обязаны объявлять ее отдельно — система модулей заполнит ее самостоятельно (в следующем разделе об этом поговорим подробно). Она ни от чего не зависит, однако экспортирует целых 116 пакетов, поэтому здесь представлена лишь урезанная версия:

```
module java.base {
 exports java.lang;
 exports java.math;
 exports java.nio;
 exports java.util;
 // больше, больше экспортов
 // остальной код опущен
}
```

Более простой модуль — *java.logging*, который включает в себя пакет `java.util.logging`:

```
module java.logging {
 exports java.util.logging;
}
```

В качестве модуля, запрашивающего другие модули, рассмотрим *java.rmi*. Он создает сообщения логов и, соответственно, зависит для этого от *java.logging*. API,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

который он выгружает, может быть найден в `java.rmi` и других пакетах с таким префиксом:

```
module java.rmi {
 requires java.logging;
 exports java.rmi;
 // экспортирование других `java.rmi.*` пакетов
 // остальной код урезан
}
```

Для получения других примеров вернитесь к разделу 2.3, особенно к коду, который объявляет модули приложений *ServiceMonitor*.

3.1.4. Множество типов модулей

Подумайте о приложении, разрабатываемом вами сейчас. Вам повезло, если оно содержит несколько JAR-файлов, которые в отдаленном будущем могут превратиться в модули. Однако приложение состоит не только из них. JDK тоже разделен на модули, и они также станут частью ваших соглашений. Но подождите, это еще не все! В данном наборе модулей часть из них имеют характеристики, благодаря которым нужно выделить их отдельно.

> **Определение: типы модулей**
>
> Во избежание неподдающегося расшифровке хаоса следующие термины идентифицируют конкретные роли и упрощают разговор о модульном ландшафте. Время остановиться и подробнее изучить их. Не переживайте, если не запомните сразу все; оставьте закладку на этой странице и вернитесь в любой момент, когда встретите термин, значение которого невозможно интерпретировать (или перейдите прямо к указателю книги).

- *Модули приложения* — модуль, не входящий в JDK; модули, которые Java-разработчики создают для собственных проектов, будь то библиотеки, фреймворки или приложения. Находятся в пути модуля. Со временем станут модульными JAR-файлами (см. подраздел 3.1.2).
- *Начальный модуль* — модуль приложения, в котором начинается компиляция (для `javac`) или содержится метод `main` (для `java`). Подраздел 5.1.1 показывает, как их разделить при запуске приложения командой `java`. Компилятор также использует эту концепцию: как описано в подразделе 4.3.5, объявляет, с какого модуля начинать компиляцию.
- *Корневые модули* — отсюда JPMS начинает распределение зависимостей (процесс, подробно описанный в подразделе 3.4.1). Вдобавок к содержанию главного класса или кода для компиляции начальный модуль также является корневым модулем. В сложных ситуациях, с которыми вы столкнетесь по мере чтения, может возникнуть необходимость определить корневые модули за пределами начального (как описано в подразделе 3.4.3).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- *Платформенные модули* — модули, из которых состоит JDK. Они объявлены платформенной спецификацией Java SE (имеют префикс *java.*) и модулями JDK (с префиксом *jdk.*). Как описано в подразделе 3.1.1, хранятся в оптимизированной форме в файле `modules` каталога выполнения `libs`.
- *Инкубационные модули* — нестандартные платформенные модули, чьи имена всегда начинаются с *jdk.incubator*. Они содержат экспериментальные API, которые может быть полезно отдать на тестирование авантюрным разработчикам, прежде чем утверждать окончательную версию.
- *Системные модули* — вдобавок к созданию образа среды выполнения из набора платформенных модулей `jlink` может также включать модули приложения. Платформенные модули и модули приложения, собранные в подобном образе, вместе называются *системными*. Чтобы получить их список, используйте команду `java` в каталоге `bin` образа и вызовите `java --list-modules`.
- *Обозреваемые модули* — все платформенные модули в текущем выполнении, как и модули приложения, определенные в командной строке; модули, которые используются JPMS для заполнения зависимостей. Собранные вместе, образуют *совокупность обозреваемых модулей*.
- *Базовый модуль* — различие между модулями приложения и платформенными модулями существует только в целях упрощения коммуникаций. Для системы модулей все модули равны, за исключением одного: платформенного *java.base*, так называемого базового модуля, играющего особенную роль.

Платформенные модули и модули приложения имеют дескрипторы, присвоенные им создателем. Существуют ли другие модули? Да:

- *явные* — платформенные модули и почти все модули приложения, которые имеют дескриптор, данный им создателем;
- *автоматические* — именованные модули без присвоенных дескрипторов (спойлер: обычные JAR-файлы в пути модуля). Это модули приложения, созданные средой выполнениея, а не разработчиком;
- *именованные* — набор явных и автоматических модулей. У них есть имена, определенные дескриптором или подразумевающиеся JPMS;
- *безымянные* — модули без имен (спойлер: содержимое пути класса) и, следовательно, не являющиеся явными.

Как автоматические, так и безымянные модули становятся актуальными в контексте миграции приложения в систему модулей — тема, подробно описанная в главе 8. Лучше понять то, как эти типы модулей взаимодействуют друг с другом, поможет рис. 3.3.

Чтобы применить указанные термины на практике, вернемся к приложению *ServiceMonitor*, исследованному в главе 2. Оно состоит из семи модулей (*monitor, monitor.observer, monitor.rest* и т. д.) плюс внешние зависимости Spark и Hibernate и их переходные зависимости.

Во время запуска папка, в которой находятся семь модулей и их зависимости, объявляется в командной строке. Вместе с платформенными модулями в JRE или

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

JDK, где запущено приложение, они формируют совокупность обозреваемых модулей. Это и есть набор модулей, из которых система модулей попытается собрать все зависимости.

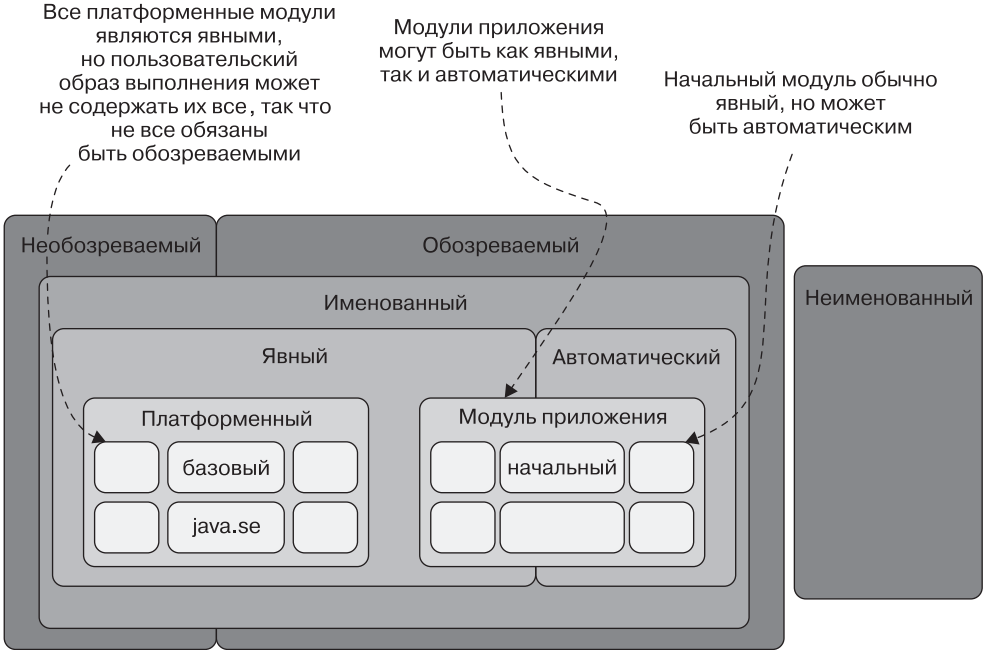


Рис. 3.3. Большинство типов модулей, организованных в простую диаграмму. Модули из JDK называются платформенными, с базовыми модулями в центре. Затем здесь присутствуют модули приложения, один из которых должен быть начальным — в нем находится метод main приложения. (Корневые, системные и инкубационные модули не показаны)

Модули *ServiceMonitor*, как и модули, составляющие их зависимости, Hibernate- и Spark-платформенные. Из-за содержания метода `main` модуль *monitor* является начальным — других корневых модулей не требуется. Единственный платформенный модуль, от которого напрямую зависит программа, — базовый модуль *java.base*, но Hibernate и Spark предоставляют и другие модули, в частности *java.sql* и *java.xml*. Поскольку это новое приложение и предполагается, что все его зависимости модуляризованы, сценарий миграции не предусмотрен; соответственно, автоматические и безымянные модули здесь не присутствуют.

Сейчас вы уже знаете существующие типы модулей и механизмы их объявления — настало время исследовать, как Java обрабатывает эту информацию.

3.2. Читабельность: соединяем части вместе

Модули — это атомарные строительные блоки: узлы в графе взаимодействующих артефактов. Но не было бы графа без ребер, соединяющих узлы! Вот здесь и появляется *читабельность*, на основе которой система модулей создаст связи между узлами.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Определение: ребро читабельности

Когда модуль *customer* запрашивает модуль *bar* из его декларации, затем во время выполнения *customer* сможет *прочитать bar* или, наоборот, *bar* будет прочитан *customer* (рис. 3.4). Связь между двумя модулями называется *ребром читабельности* или, для краткости, *ребром чтения*.

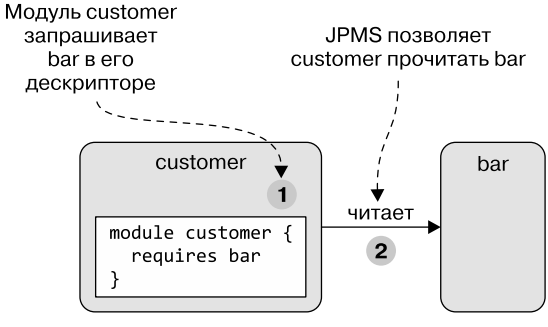


Рис. 3.4. Модуль customer запрашивает модуль bar в его дескрипторе *(1)*. На основании этого система модулей позволит customer прочитать bar во время выполнения *(2)*

Принимая во внимание, что такие фразы, как «*customer* запрашивает *bar*» или «*customer* зависит от *bar*», отражают статические отношения во время компиляции между *customer* и *bar*, читабельность является более динамичным аналогом во время выполнения. Почему более динамичным? Директива `requires` — первичный источник ребер чтения, но ни в коем случае не единственный. Существуют также опции командной строки (см. `--add-reads` в подразделе 3.4.4) и API рефлексии (см. подраздел 12.3.4), и оба варианта могут быть использованы для добавления новых ребер; в конце концов, это не имеет значения. Вне зависимости от того, как можно прочитать ребра, результат будет примерно одинаковым: они станут основой для надежной настройки и доступности (см. раздел 3.3).

3.2.1. Достижение надежной настройки

Как описано в подразделе 1.5.1, надежная настройка направлена на достижение того, чтобы определенная конфигурация артефактов, с которыми Java-программа компилируется или запускается, могла поддерживать эту программу без ложных ошибок во время выполнения. С этой целью она проводит пару проверок (процесс во время разбора модуля описан в подразделе 3.4.1).

ВАЖНАЯ ИНФОРМАЦИЯ

Система модулей проверяет, содержит ли совокупность обозреваемых модулей все необходимые зависимости, прямые и переходные, и выдает ошибку, если что-то пропущено. Здесь не может быть двусмысленности: никакие

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

два артефакта не могут утверждать, что являются одним и тем же модулем. Это особенно интересно в случае, когда представлены две версии одного и того же модуля, — ввиду того, что система модулей не признает концепции версионности (см. главу 13) и рассматривает их как дублирующийся модуль. Соответственно, она выдаст ошибку в подобной ситуации. Между модулями не может быть статических циклов зависимости. Во время выполнения возможно и иногда даже необходимо, чтобы модули могли иметь доступ друг к другу (вспомните код, использующий аннотации и рефлексии Spring одновременно). Однако не может быть зависимостей при компиляции (очевидно, Spring не компилируется с кодом, который он рефлексирует). Пакеты должны иметь уникальный источник, так что два модуля не могут содержать одни и те же типы в одном пакете. Если они это делают, то подобное явление называется разделенным пакетом, и система модулей откажется компилировать или запускать такие настройки. Это особенно интересно в контексте миграции, поскольку некоторые существующие библиотеки и фреймворки намеренно разделяют пакеты (см. раздел 7.2).

Такая проверка, конечно, не герметична, и некоторые проблемы могут скрываться достаточно долго, прежде чем сломать работающее приложение. Если, например, неправильная версия модуля находится в правильном месте, то приложение запустится (все требуемые модули присутствуют), но упадет позже, когда, скажем, обнаружится отсутствие класса или метода.

Поскольку система модулей разработана, чтобы обеспечить согласованное поведение во время компиляции и выполнения, эти ошибки со временем минимизируются, если компиляция и выполнение базируются на одних и тех же артефактах. (В примере компиляция модуля с неправильной версией провалилась бы.)

3.2.2. Эксперименты с ненадежными настройками

Попробуем что-нибудь сломать! Какие из настроек система модулей определила как ненадежные? Исследуем это, вернувшись к приложению *ServiceMonitor*, введенному в главе 2.

Пропущенные зависимости

Рассмотрим *monitor.observer.alpha* и его декларацию:

```
module monitor.observer.alpha {
 requires monitor.observer;
 exports monitor.observer.alpha;
}
```

Вот как будет выглядеть попытка скомпилировать его с отсутствующим *monitor.observer*:

```
> monitor.observer.alpha/src/main/java/module-info.java:2:
>     error: module not found: monitor.observer
>         requires monitor.observer
>                          ^
> 1 error
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Если модуль присутствует во время компиляции, но пропадает на пути к запуску, то JVM завершит работу со следующей ошибкой:

```
> Error occurred during initialization of boot layer
> java.lang.module.FindException:
>     Module monitor.observer not found,
>     required by monitor.observer.alpha
```

Хотя имеет смысл принудительно проверять наличие всех переходно-требуемых модулей во время запуска, то же самое не подходит для компилятора. Соответственно, если непрямые зависимости пропущены, то компилятор не выдаст ни предупреждения, ни ошибки, в чем можно убедиться на следующем примере.

Это декларации модулей *monitor.persistence* и *monitor.statistics*:

```
module monitor.persistence {
 requires monitor.statistics;
 exports monitor.persistence;
}

module monitor.statistics {
 requires monitor.observer;
 exports monitor.statistics;
}
```

Очевидно, что *monitor.persistence* не запрашивает *monitor.observer* напрямую, поэтому компиляция *monitor.persistence* пройдет успешно даже в случае отсутствия *monitor.observer* в пути модуля.

Запуск приложения с пропущенными переходными зависимостями не сработает. Даже если начальный модуль не зависит от них напрямую, это не касается некоторых других модулей, так что они отметят их как пропущенные. Ветка `break-missing-transitive-dependency` в репозитории *ServiceMonitor* создает настройку, в которой пропущенный модуль приведет к сообщению об ошибке.

Дублирующиеся модули

Поскольку модули ссылаются друг на друга с помощью имен, ситуация, когда два разных модуля имеют одинаковые имена, двусмысленна. Какой из них нужно выбрать, напрямую зависит от контекста, и это не то решение, которое система модулей может принять самостоятельно. Как следствие, вместо принятия потенциально неверного решения она не принимает никакое, выдавая ошибку. Такой немедленный сбой позволяет разработчикам тут же заметить проблему и исправить ее, прежде чем она приведет к еще более разрушительным последствиям.

Это ошибка компиляции, которую выдала система модулей после попытки скомпилировать модуль с двумя вариантами *monitor.observer.beta* в пути модуля:

```
> error: duplicate module on application module path
>     module in monitor.observer.beta
> 1 error
```

Обратите внимание: компилятор не может связать ошибку ни с одним из компилируемых файлов, поскольку не они являются причиной проблемы. Вместо этого ошибку на пути модуля вызывают артефакты.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Когда ошибка проходит незаметно до момента запуска JVM, появится более конкретизированное сообщение, которое перечисляет имена файлов:

```
> Error occurred during initialization of boot layer
> java.lang.module.FindException:
>     Two versions of module monitor.observer.beta found in mods
>     (monitor.observer.beta.jar and monitor.observer.gamma.jar)
```

Как мы уже обсудили в подразделе 1.5 и еще будем рассматривать в разделе 13.1, система модулей не имеет концепции версионности, поэтому в данном случае выведется та же самая ошибка. Я бы сказал так: хорошей догадкой является то, что подавляющее большинство ошибок при дублировании модулей будет вызвано наличием одного и того же модуля в нескольких версиях в пути модуля.

ВАЖНАЯ ИНФОРМАЦИЯ

Проверки на двусмысленность применяются только к отдельным записям в пути модуля! (Данное предложение может вас запутать — я объясню, что имел в виду, в подразделе 3.4.1, но хотел бы упомянуть это здесь, чтобы данный важный факт не остался незамеченным.)

Кроме того, система модулей выдаст ошибку о дублировании модулей, если модуль на самом деле необязателен. Достаточно того, что он *присутствует* в пути модуля! Тому есть две причины: сервисы и необязательные зависимости, которые представлены в главе 10 и в разделе 11.2. Ветка `break-duplicate-modules-even-if-unrequired` приложения *ServiceMonitor* создает сообщение об ошибке дублирования несмотря на то что этот модуль необязателен.

Циклы зависимостей

Случайно создать цикличную зависимость просто, сложно провести ее мимо компилятора. Непросто даже *представить* ее компилятору. Для этого вам придется решить проблему курицы и яйца: когда два проекта зависят друг от друга, невозможно скомпилировать один прежде другого. При попытке сделать это вы получите пропущенные зависимости и соответствующие ошибки.

Единственный способ скомпилировать оба модуля сразу — если можно так выразиться, начать с курицы и яйца одновременно: раздел 4.3 объясняет, как это сделать. Достаточно сказать, что если попробовать скомпилировать циклическую зависимость между модулями, то система модулей распознает это и выдаст ошибку компиляции. Вот как выглядит описанное, если *monitor.persistence* и *monitor.statistics* зависят друг от друга:

```
> monitor.statistics/src/main/java/module-info.java:3:
>     error: cyclic dependence involving monitor.persistence
>         requires monitor.persistence;
>                  ^
> 1 error
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Другой способ добиться этого — установить циклическую зависимость не сразу, а спустя время *после* того, как правильная настройка будет собрана. Еще один пример с *monitor.persistence* и *monitor.statistics*:

```
module monitor.persistence {
 requires monitor.statistics;
 exports monitor.persistence;
}

module monitor.statistics {
 requires monitor.observer;
 exports monitor.statistics;
}
```

Данные настройки правильны и компилируются без проблем. Теперь начинается хитрость: скомпилируйте модули и сохраните JAR-файлы. Затем измените декларацию модуля *monitor.statistics* так, чтобы она запрашивала *monitor.persistence*, что создаст циклическую зависимость (в данном примере изменение не имеет смысла, в отличие от многих больших приложений):

```
module monitor.statistics {
 requires monitor.observer;
 requires monitor.persistence;
 exports monitor.statistics;
}
```

Следующий шаг — скомпилировать только измененный *monitor.statistics* с уже скомпилированными модулями в пути модуля. Они должны включать в себя *monitor.persistence*, поскольку модуль статистики сейчас зависит от него. В свою очередь, модуль состояния все еще объявляет свою зависимость от *monitor.statistics* — это вторая половина цикла зависимости. К сожалению, данный раунд взлома система модулей распознает и выдаст ту же ошибку, что и раньше.

Перевод этой «игры в наперстки» на следующий уровень наконец обманывает компилятор. По данному сценарию два никак не связанных модуля — выберем *monitor.persistence* и *monitor.rest* — компилируются в модульные JAR-файлы.

Одну зависимость нужно добавить, скажем *persistence* от *rest*, и измененный *persistence* следует скомпилировать с оригинальным набором модулей. Это сработает, поскольку оригинальный *rest* не зависит от *persistence*.

Вторая зависимость, *rest* от *persistence*, также добавлена, но и *rest* компилируется с оригинальным набором модулей, включая версию *persistence*, которая еще от него не зависит. Как следствие, все это тоже отлично скомпилируется.

Удивлены? Взгляните на рис. 3.5, чтобы получить другое объяснение.

В итоге у нас есть версии *monitor.persistence* и *monitor.rest*, которые зависят друг от друга. Чтобы так произошло в реальной жизни, процесс компиляции — возможно, управляемый инструментом сборки — должен находиться в серьезном беспорядке (но это не случайно). К счастью, система модулей прикроет вас и вернет ошибку, когда JVM запустится с подобной настройкой:

```
> Error occurred during initialization of boot layer
> java.lang.module.FindException:
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
>     Cycle detected:
>         monitor.persistence
>           -> monitor.rest
>           -> monitor.persistence
```

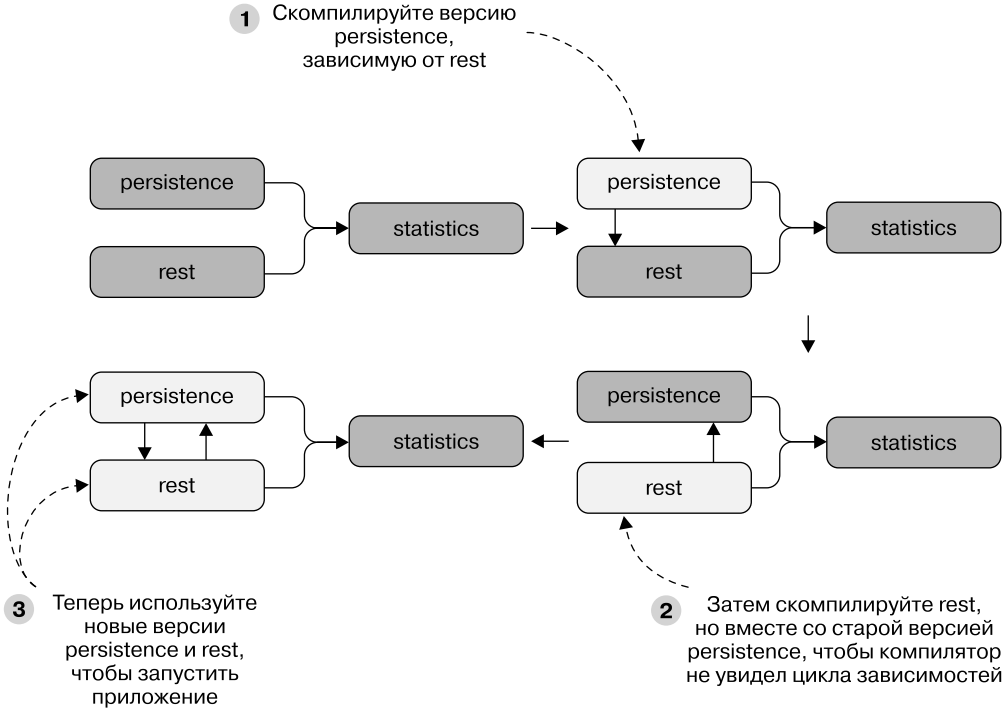


Рис. 3.5. Провести циклы зависимостей мимо компилятора нелегко. Здесь же это сделано с помощью двух независимых модулей, persistence и rest (оба зависят от statistics), и затем поочередного добавления зависимостей от одного к другому. Важно компилировать rest со старым persistence, чтобы цикл не был заметен и компиляция прошла успешно. В конце оба оригинальных модуля заменяются на только что скомпилированные версии, которые имеют циклическую зависимость друг от друга

Все примеры показывают цикличную зависимость между двумя артефактами, но система модулей распознает циклы любой длины. И хорошо, что она это делает! Изменения всегда рискуют нарушить работоспособность исходного кода, то есть другого кода, который использует изменяемый код — напрямую или транзитивно.

Если зависимости идут в одном направлении, то появляется очень много кода, на который могут повлиять изменения. С другой стороны, если зависимости могут сформировать цикл, то это может затронуть весь код в таком цикле и все от него зависящее. Особенно это касается больших циклов, когда может получиться так, что *весь* код попадет под влияние, и я уверен в вашем желании этого избежать. И система модулей не одинока в стремлении вам помочь — инструмент сборки тоже набросится на циклы зависимостей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Разделенные пакеты

Такие пакеты появляются, когда два модуля содержат типы в одном и том же пакете. Например, вспомните, что модуль *monitor.statistics* содержит класс `Statistician` в пакете `monitor.statistics`. Теперь же представим, что в модуль *monitor* входит простая резервная реализация, `SimpleStatistician`, и, чтобы поддержать единообразие, она находится в собственном пакете модуля *monitor*, `monitor.statistics`.

При попытке скомпилировать *monitor* вы получите следующую ошибку:

```
> monitor/src/main/java/monitor/statistics/SimpleStatistician.java:1:
>     error: package exists in another module: monitor.statistics
>         package monitor.statistics;
>         ^
> 1 error
```

ВАЖНАЯ ИНФОРМАЦИЯ

Интересно, что компилятор показывает ошибку только в том случае, если компилируемый модуль имеет доступ к разделенному пакету другого модуля. Это значит, разделенный пакет должен быть экспортирован.

Чтобы это воспроизвести, пойдем другим путем: `SimpleStatistician` прошел, и настало время модуля *monitor.statistics*, который создаст разделенный пакет. Пытаясь использовать некоторые методы утилит, он создает класс `Utils` в пакете `monitor`. Он не хочет делить данный класс с другими модулями, поэтому продолжает экспортировать только пакет `monitor.statistics`.

Компиляция модуля *monitor.statistics* прошла без ошибок, что верно, поскольку он не запрашивает *monitor* и потому не имеет представления о разделенном пакете. Самое интересное начинается, когда наступает время компиляции модуля *monitor*. Он зависит от *monitor.statistics* и к тому же содержит типы в пакете `monitor`. Но, как я уже упомянул, ввиду того, что *monitor.statistics* не экспортирует пакет, компиляция завершается успешно.

Прекрасно! Пришло время запуска:

```
> Error occurred during initialization of boot layer
> java.lang.reflect.LayerInstantiationException:
>     Package monitor in both module monitor.statistics and module monitor
```

Что-то пошло не так. Система модулей проверяет наличие разделенных пакетов при запуске, и здесь не имеет значения, экспортируются ли они: два модуля не могут содержать типы в одинаковом пакете. Как покажет раздел 7.2, это может перерасти в проблему с миграцией кода на Java 9.

Репозиторий *ServiceMonitor* демонстрирует проблему разделенных пакетов во время компиляции и выполнения в ветках `break-split-package-compilation` и `break-split-package-launch`.

Модульный алмаз смерти

Особенно коварная смесь разделенных пакетов и отсутствующих зависимостей — модульный алмаз смерти (рис. 3.6). Представим, что модуль был переименован при переходе от одного выпуска к другому: одна из зависимостей запрашивает его по

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

старому имени, а другая — по новому. Теперь нужно, чтобы под разными именами хранился один и тот же код, но JPMS не позволяет этому случиться.

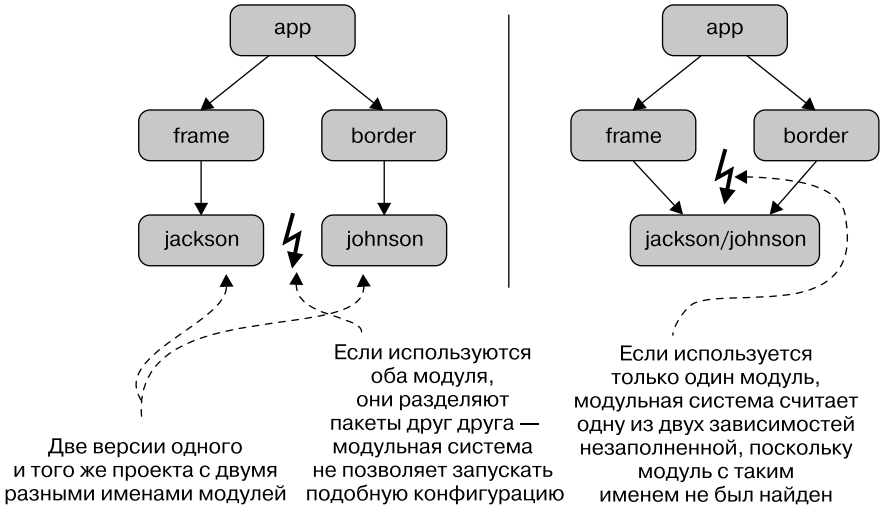


Рис. 3.6. Если модуль меняет имя (здесь jackson на johnson), то проекты, которые зависят от него дважды (здесь app через frame и border), могут в конечном итоге столкнуться с модульным алмазом смерти: они зависят от одного проекта, но с разными именами

Получится одна из двух ситуаций:

- один модульный JAR, который может появиться только как модуль с одним именем, и это вызовет ошибку, поскольку одна из зависимостей не сможет быть заполнена;
- два модульных JAR с разными именами, но одинаковыми пакетами, что станет причиной ошибки разделенных пакетов, которая рассматривалась выше.

ВАЖНАЯ ИНФОРМАЦИЯ

Вы должны избегать таких ситуаций любой ценой! При загрузке артефактов в публичный репозиторий следует внимательно проверить, необходимо ли переименовывать модуль. Если да, то нужно будет изменить и имена пакетов, чтобы люди могли применять старые и новые модули бок о бок. Если вы окажетесь в этой ситуации как пользователь, то вам может повезти, если вы можете создать модуль агрегатора (см. подраздел 11.1.5) или отредактировать дескрипторы модуля (см. подраздел 9.3.3).

3.3. Доступность: объявление публичных API

Познакомившись с модулями и ребрами читабельности, теперь вы знаете, как система модулей строит планируемый вами граф. Чтобы он не вел себя как комок грязи, от которого хотелось бы избавиться, есть еще одно требование: возможность скрывать внутренние компоненты модуля с целью исключить получение любым внешним кодом доступа к нему. Здесь и появляется *доступность*.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Определение: доступность

Тип `Drink` в модуле *bar* доступен коду в модуле *customer*, если выполняются все нижеперечисленные условия (рис. 3.7):

- `Drink` — публичный тип;
- `Drink` принадлежит пакету, который экспортируется *bar*;
- *customer* читает *bar*.

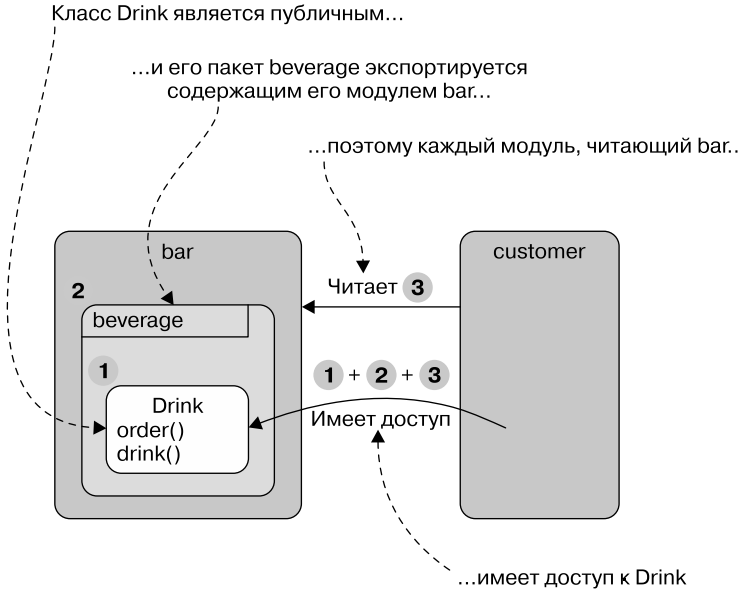


Рис. 3.7. Модуль bar содержит публичный тип Drink *(1)* в экспортируемом пакете *(2)*. Модуль customer читает модуль bar *(3)*, поэтому все условия для доступа кода в customer к Drink выполняются. Интересно, что произойдет, если они не выполнятся? Обратитесь к подразделу 3.3.3

Для членов доступного типа (то есть его полей, методов и вложенных классов) действуют обычные правила видимости: открытые члены полностью доступны, а защищенные — только для классов-наследников. Технически закрытые члены пакета доступны в одном и том же пакете, но, как показал предыдущий раздел, это бесполезно из-за правила, запрещающего разделение пакетов между модулями.

ПРИМЕЧАНИЕ

Определение доступности включает в себя модуль, который хочет получить доступ к типу. В этом смысле тип никогда не является «доступным», а только «доступным для определенного модуля». Однако обычно используется один и тот же термин, даже если не видно другого модуля, и говорят, что тип доступен, если публичен и находится в экспортированном пакете. Любой модуль может свободно получить доступ к типу, прочитав модуль, содержащий его.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Для уяснения того, как доступность влияет на публичные API модуля, важно прежде всего понимать следующий термин: что такое *публичный API*?

Определение: публичный API

В нетехнических терминах *публичный API* модуля — это все, что нельзя изменить, не вызывая ошибок компиляции в коде, который его использует. (Как правило, термин также включает в себя спецификацию поведения во время выполнения, но, поскольку система модулей не работает в этом измерении, я проигнорирую это в данной книге.) С технической точки зрения публичный API модуля состоит из следующих элементов:

- имен всех публичных типов в экспортируемых пакетах;
- имен собственных и имен типов публичных и защищенных полей;
- имен собственных, имен типов аргументов и имен возвращаемых типов всех публичных и защищенных методов (называемых *сигнатурами методов*).

Если вам покажется странным мое внезапное упоминание имен, то подумайте о том, что вы можете изменить в типе, сохраняя зависимый код вне компиляции пакета. Закрытые и видимые в пакете поля? Определенно! Приватные и видимые для пакета методы? Конечно. Тела публичных методов? Да. Единственное, что не нужно трогать, — это имена всего, с чем может быть скомпилирован другой код: имя типа, сигнатуры публичных методов и т. д.

Рассматривая определение того, что составляет публичный API, мы понимаем: система модулей меняет все, начиная с Java 9, на уровне пакетов (должны быть экспортированы) и типов (должны быть публичными). С другой стороны, в типе ничего не изменилось и публичный API типа в Java 8 такой же, как в Java 9 и более поздних версиях.

3.3.1. Обеспечение надежной инкапсуляции

ВАЖНАЯ ИНФОРМАЦИЯ

Если тип недоступен, то невозможно взаимодействовать с ним каким-либо образом, специфическим для этого типа: нельзя создать его экземпляр, получить доступ к его полям, вызвать методы или использовать вложенные классы. Фраза «специфический для этого типа» немного необычна — что она означает? Возможность взаимодействовать с членами типа, если они определены в доступном супертипе, например в интерфейсе, который реализует тип, или, в конце концов, в Object. Это очень похоже на Java 9, где можно было применить непубличную реализацию публичного интерфейса, но только через данный интерфейс.

В качестве примера рассмотрим высокопроизводительную библиотеку *superfast* с пользовательскими реализациями известных коллекций Java. Сосредоточимся на гипотетическом классе `SuperfastHashMap`, который реализует Java-интерфейс `Map` и недоступен (возможно, этот пакет видим в экспортированном пакете или вообще не экспортируется).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Если код вне модуля *superfast* получает экземпляр `SuperfastHashMap` (возможно, из фабрики), то ограничивается использованием его в качестве `Map`. Он не может ни присвоить его переменной типа `SuperfastHashMap`, ни вызвать для него `superfastGet` (даже если это публичный метод). Но все, что определено в доступных супертипах, таких как `Map` и `Object`, не представляет проблемы (рис. 3.8).

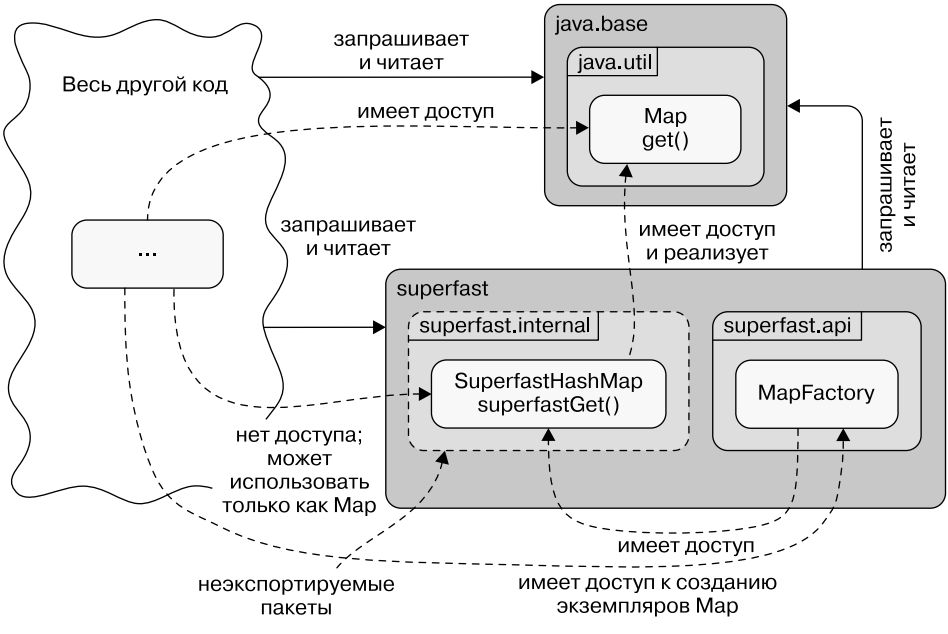


Рис. 3.8. Недоступный тип SuperfastHashMap реализует доступный интерфейс Map. Код вне модуля superfast, получив экземпляр, может использовать его как Map или Object, но никогда способами, специфическими для этого типа: например, путем вызова superfastGet. Код в модуле superfast не ограничен доступностью и может применять тип как обычно: например, для создания экземпляров и их возврата

Правила доступности позволяют выгружать тщательно отобранные функции, в то же время строго инкапсулируя внутренние компоненты модуля, проверяя независимость внешнего кода от деталей реализации. Интересно, что это включает в себя рефлексию, которая не может обойти правила, даже если используется за пределами модуля! (Мы поговорим о ней в оставшейся части главы. Если вам нужно разобраться в основах, см. приложение Б.)

Может быть, вам интересно, как библиотеки на основе рефлексии, такие как Spring, Guice, Hibernate и другие, станут работать в будущем или как код может быть разобранным на модули в случае необходимости. Есть несколько способов дать или получить доступ:

- обычный экспорт (см. раздел 3.1);
- квалифицированный экспорт (см. раздел 11.3);
- открытые модули и открытые пакеты (см. раздел 12.2);
- параметры командной строки (собраны в разделе 7.1).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Глава 12 исследует рефлексию более подробно.

Но вернемся к трем условиям — предпосылке доступности (публичный тип, экспортированный пакет, чтение модуля). Они имеют интересные последствия.

ВАЖНАЯ ИНФОРМАЦИЯ

Например, public больше не является публичным. По типу больше невозможно определить, будет ли он виден за пределами модуля, — для этого необходимо проверить module-info.java или довериться IDE, чтобы та подсветила экспортированные пакеты или типы. Без директив requires все типы в модуле недоступны извне. Инкапсуляция — новое значение по умолчанию!

Кроме того, три условия подразумевают, что теперь невозможно случайно положиться на переходные зависимости. Разберемся почему.

3.3.2. Инкапсуляция переходных зависимостей

Без системы модулей можно применять типы из JAR, в которые входит зависимость, но это не объявляется как зависимость. Как только проект использует типы таким образом, конфигурация сборки больше не отражает истинный набор зависимостей, что может привести к чему угодно: от некорректных архитектурных решений до ошибок времени выполнения.

В качестве примера предположим, что проект использует Spring, зависящий от OkHttp. Написать код, использующий типы из OkHttp, так же просто, как позволить IDE добавить операторы импорта, которые она предлагает. Код будет скомпилирован и запущен, поскольку инструмент сборки гарантирует постоянное присутствие Spring и всех его зависимостей, включая OkHttp. Это делает необязательным объявление зависимости от OkHttp, так что легко забыть о ней (рис. 3.9).

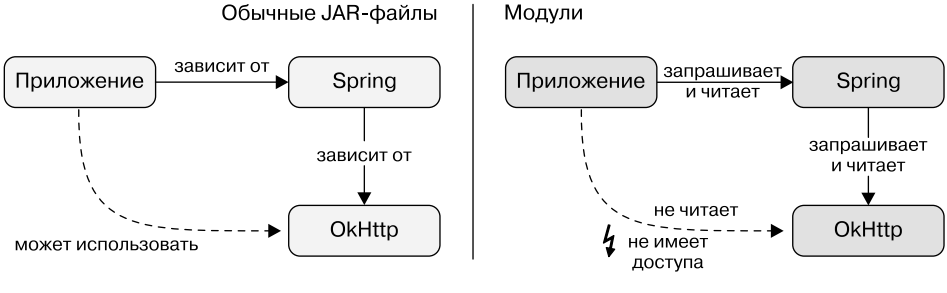


Рис. 3.9. Без модулей легко случайно получить связь с переходными зависимостями, как в этом примере, где приложение зависит от OkHttp, который загружается через Spring. С другой стороны, при наличии модулей зависимости должны быть объявлены с помощью директивы requires, чтобы к ним можно было получить доступ. Приложение не запрашивает OkHttp и поэтому не может получить доступ к нему

Как следствие, анализ зависимостей проекта даст ошибочные результаты, на основе которых могут быть приняты проблемные решения. Версия OkHttp также не исправлена и полностью зависит от того, что используется в Spring. Если эта версия обновляется, то код, зависящий от OkHttp, автоматически работает на другой

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

версии, создавая реальный риск того, что программа будет давать сбои или аварийно завершится во время выполнения.

Из-за того, что система модулей требует, чтобы модуль доступа прочитал модуль, к которому осуществляется доступ, этого больше не произойдет. Если проект не объявит свою зависимость от OkHttp с помощью директивы `requires`, то система модулей не позволит ему получить доступ к классам OkHttp. Таким образом, это заставит вас поддерживать свои конфигурации в актуальном состоянии.

Обратите внимание: модули имеют возможность передавать собственные зависимости модулям, которые зависят от них, с помощью функции, называемой *подразумеваемой читабельностью*. Подробности см. в разделе 11.1.

3.3.3. Столкновения инкапсуляций

Пора что-нибудь сломать, как мы сделали с читабельностью! Но прежде я хочу показать сценарий, который следует всем правилам — и работает. И снова он использует приложение *ServiceMonitor*, представленное в главе 2.

Ради этих примеров предположим, что модуль *monitor.observer* содержал в своем пакете `monitor.observer` класс `DisconnectedServiceObserver`. Его действия не имеют значения: важно то, что он реализует `ServiceObserver`, имеет конструктор, который не требует никаких аргументов, и то, что модуль *monitor* использует его.

Модуль *monitor.observer* экспортирует `monitor.observer`, а `DisconnectedServiceObserver` является публичным. Это удовлетворяет первым двум требованиям доступности, вследствие чего другие модули могут получить к нему доступ, если прочитают *monitor.observer*. Модуль *monitor* также выполняет это предварительное условие, поскольку запрашивает *module.observer* в декларации модуля. Взятые вместе (рис. 3.10 и листинг 3.1), требования выполнены, и код в мониторе может получить доступ к `DisconnectedServiceObserver`. Соответственно, компиляция и выполнение проходят без ошибок. Уделим особое внимание подробностям и посмотрим, как реагирует система модулей.

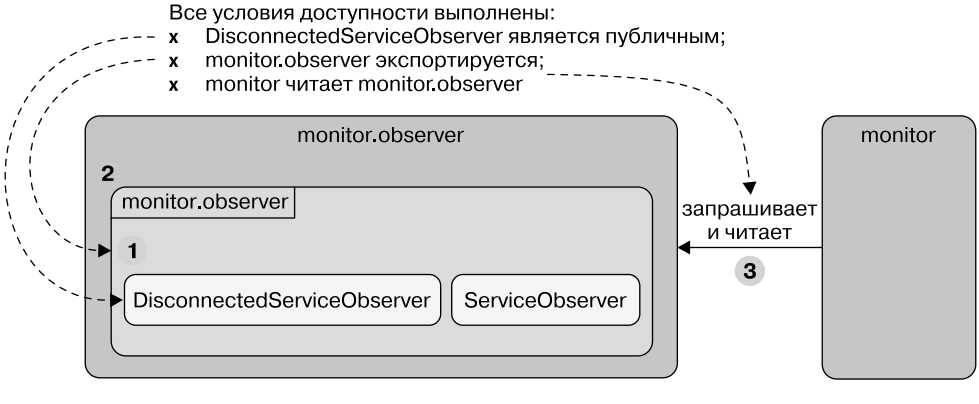


Рис. 3.10. DisconnectedServiceObserver является публичным *(1)* и находится в пакете, экспортированном monitor.observer *(2)*. Поскольку модуль monitor читает monitor.observer *(3)*, код в нем может использовать DisconnectedServiceObserver

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Листинг 3.1 DisconnectedServiceObserver, доступный модулю monitor

```
// --- TYPE DisconnectedServiceObserver ---
package monitor.observer;

public class DisconnectedServiceObserver
  implements ServiceObserver {
 // тело класса опущено
}

// --- MODULE DECLARATION monitor.observer ---
module monitor.observer {
 exports monitor.observer;
}

// --- MODULE DECLARATION monitor ---
module monitor {
 requires monitor.observer;
 // другие директивы requires опущены
}
```

Публичный monitor.observer.
DisconnectedServiceObserver

Модуль monitor.observer экспортирует
пакет monitor.observer

Модуль monitor запрашивает
и в итоге читает monitor.observer

Тип не публичный

Если `DisconnectedServiceObserver` сделан видимым в пакете, то компиляция модуля *monitor* завершается неудачно. Точнее, импорт выведет первую ошибку:

```
> monitor/src/main/java/monitor/Monitor.java:4: error:
>     DisconnectedServiceObserver is not public in monitor.observer;
>     cannot be accessed from outside package
> import monitor.observer.DisconnectedServiceObserver;
>
```

Доступ к типам, видимым в пакете, из другого пакета также не был возможен до Java 9, и по этой причине сообщение об ошибке не является чем-то новым — вы получите такое же без работающей системы модулей.

Аналогично если вы обойдете проверки компилятора, перекомпилировав только модуль `monitor.observer` после того, как `DisconnectedServiceObserver` станет видимым в пакете, а затем запустите все приложение, то ошибка будет такой же, как и без системы модулей:

```
> Exception in thread "main" java.lang.IllegalAccessError:
>     failed to access class monitor.observer.DisconnectedServiceObserver
>     from class monitor.Monitor
```

До Java 9 можно было использовать API рефлексии для доступа к типу во время выполнения, и это то, чему препятствует надежная инкапсуляция. Рассмотрим следующий код:

```
Constructor<?> constructor = Class
 .forName("monitor.observer.DisconnectedServiceObserver")
 .getDeclaredConstructor();
constructor.setAccessible(true);
ServiceObserver observer = (ServiceObserver) constructor.newInstance();
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В Java 8 и ранее это работает независимо от того, является ли `DisconnectedServiceObserver` публичным или видимым в пакете. В Java 9 и более поздних версиях система модулей запрещает доступ, если `DisconnectedServiceObserver` является видимым в пакете и вызов `setAccessible` становится причиной ошибки:

```
> Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
>     Unable to make monitor.observer.DisconnectedServiceObserver()
>     accessible: module monitor.observer does not "opens monitor.observer"
>     to module monitor
```

Ветка `break-reflection-over-internals` репозитория *ServiceMonitor* демонстрирует поведение, показанное здесь. Жалоба на то, что *monitor.observer* не открывает *monitor.observer*, указывает на единственное решение данной проблемы — то, что рассматривается в разделе 12.2.

Пакеты не экспортируются

Следующее в списке требований — пакет, содержащий доступный тип, должен быть экспортирован. Чтобы поэкспериментировать с этим, снова сделаем `DisconnectedServiceObserver` публичным, но переместим его в другой пакет *monitor.observer.dis*, который *monitor.observer* не экспортирует. Импорт в *monitor* обновляется до нового пакета:

```
> monitor/src/main/java/monitor/Monitor.java:4: error:
>     package monitor.observer.dis does not exist
> import monitor.observer.dis.DisconnectedServiceObserver;
>                            ^
> (package monitor.observer.dis is declared in module
>  monitor.observer, which does not export it)
```

Это достаточно просто.

Увидеть происходящее со временем выполнения в данном случае можно, если обойти проверки компилятора. Для этого отредактируйте *monitor.observer*, чтобы экспортировать *monitor.observer.dis*, скомпилируйте все модули, а затем снова скомпилируйте *monitor.observer* без данного экспорта. Вы можете запустить приложение как прежде, и получить ошибку выполнения:

```
> Exception in thread "main" java.lang.IllegalAccessError:
>     class monitor.Monitor (in module monitor) cannot access class
>     monitor.observer.dis.DisconnectedServiceObserver (in module
>     monitor.observer) because module monitor.observer does not export
>     monitor.observer.dis to module monitor
```

Подобно компилятору, среда выполнения довольно «разговорчива» и объясняет, в чем заключается проблема. То же самое относится и к API рефлексии в случае попытки сделать конструктор доступным, поэтому можно создать экземпляр `DisconnectedServiceObserver`:

```
> Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
>     Unable to make public
>     monitor.observer.dis.DisconnectedServiceObserver() accessible:
>     module monitor.observer does not "exports monitor.observer.dis"
>     to module monitor
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Если посмотреть внимательно, то можно увидеть, что и среда выполнения, и API рефлексии говорят об экспорте пакета *в* модуль. Это называется *квалифицированным экспортом* (описывается в разделе 11.3).

Модуль не читается

Последнее требование в списке состоит в том, что экспортирующий модуль должен быть прочитан тем, кто обращается к типу. Удаление директивы `requires monitor.observer` из декларации модуля *monitor* приводит к ожидаемой ошибке во время компиляции:

```
> monitor/src/main/java/monitor/Monitor.java:3: error:
>     package monitor.observer is not visible
> import monitor.observer.DiagnosticDataPoint;
>                ^
> (package monitor.observer is declared in module
>  monitor.observer, but module monitor does not read it)
```

Чтобы увидеть, как среда выполнения реагирует на отсутствующую директиву `requires`, сначала скомпилируйте все приложение с рабочей конфигурацией, которая означает, что *monitor* читает *monitor.observer*. Затем удалите директиву `requires` из файла `module-info.java` модуля *monitor* и перекомпилируйте только этот файл. Таким образом, код модуля компилируется с декларацией модуля, для которой все еще требуется *monitor.observer*, но среда выполнения увидит описание модуля, не требующее ничего подобного. Как и ожидалось, в результате появится ошибка выполнения:

```
> Exception in thread "main" java.lang.IllegalAccessError:
>     class monitor.Monitor (in module monitor) cannot access class
>     monitor.observer.DisconnectedServiceObserver (in module
>     monitor.observer) because module monitor does not read module
>     monitor.observer
```

И снова сообщение об ошибке довольно однозначное.

Наконец, перейдем к рефлексии. Вы можете использовать тот же прием компиляции, чтобы создать модуль *monitor*, который не читает *monitor.observer*. Можно также повторно применить код рефлексии из примера выше, когда `DisconnectedServiceObserver` не был публичным, но вы все равно хотели создать экземпляр класса.

Я догадываюсь, что совместный запуск этих модулей тоже выдает ошибку, верно? Да, но не такую, какую можно было ожидать:

```
> Exception in thread "main" java.lang.IllegalAccessError:
>     class monitor.Monitor (in module monitor) cannot access class
>     monitor.observer.ServiceObserver (in module monitor.observer)
>     because module monitor does not read module monitor.observer
```

Почему сообщение об ошибке жалуется на `ServiceObserver`? Потому что этот тип тоже находится в *monitor.observer*, который *monitor* больше не читает. Изменим код рефлексии, чтобы использовать только `Object`:

```
Constructor<?> constructor = Class
 .forName("monitor.observer.DisconnectedServiceObserver")
 .getDeclaredConstructor();
constructor.setAccessible(true);
Object observer = constructor.newInstance();
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Запустите это — все работает! Но как насчет отсутствующего ребра читабельности? Ответ прост, но поначалу слегка удивляет: API рефлексии заполняет его автоматически. Подраздел 12.3.1 исследует причины этого.

3.4. Путь модуля: позвольте Java узнать о модулях

Вы уже знаете порядок определения модулей и их основных свойств. Остается неясным вот что: как сообщить о них компилятору и среде выполнения. В главе 4 рассматривается создание модулей от исходного кода до JAR, и вы быстро столкнетесь с ситуацией, когда нужно сослаться на существующие модули, от которых зависит компилируемый код. Ситуация та же, что и в главе 5, где среда выполнения должна знать о модулях приложения, чтобы запустить один из них.

До Java 9 использовался бы путь класса, на котором были простые JAR-файлы (в приложении A сделаны выводы о пути класса), чтобы сообщить компилятору и среде выполнения, где искать артефакты. Они ищут его во время поиска отдельных типов, требуемых во время компиляции или выполнения.

С другой стороны, система модулей обещает не оперировать типами, а подняться на один уровень выше и вместо этого управлять модулями. Один из способов выражения этого подхода — новая концепция, которая параллельна пути класса, но использует модули вместо простых типов или JAR-файлов.

> **Определение: путь модуля**
>
> *Путь модуля* — это список, элементами которого являются артефакты или каталоги, содержащие артефакты. В зависимости от ОС элементы пути модуля разделяются двоеточием (на основе Unix) или точкой с запятой (Windows). Система модулей использует путь для поиска необходимых модулей, которые не найдены среди платформенных модулей. Обрабатывать его могут и `javac`, и `java`, а также другие связанные с модулем команды — параметры командной строки: `--module-path` и `-p`.

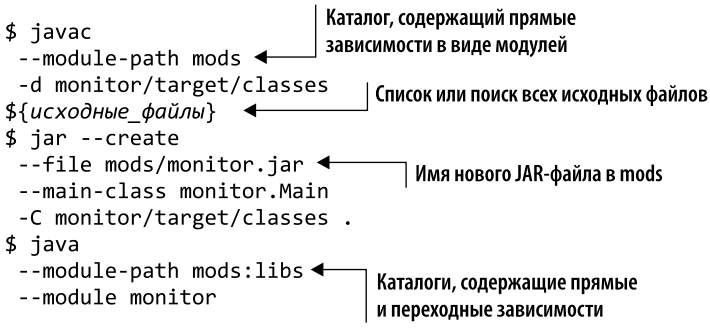
В листинге 3.2 показано, как модуль *monitor* приложения *ServiceMonitor* мог быть составлен, упакован и запущен. Он использует `--module-path`, чтобы указать на каталог `mods`, который предположительно содержит все необходимые зависимости в виде модульных JAR-файлов. Более подробную информацию о компиляции, упаковке и запуске см. в разделах 4.2, 4.5 и 5.1.

ВАЖНАЯ ИНФОРМАЦИЯ

Важно уточнить: только путь модуля обрабатывает артефакты как модули. Вооружившись данным знанием, вы можете быть немного более точными относительно того, что составляет совокупность обозреваемых модулей. В подразделе 3.1.4 это определено следующим образом: все платформенные модули в текущей среде выполнения, а также все модули приложения, указанные в командной строке, называются обозреваемыми, и вместе они составляют совокупность обозреваемых модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Листинг 3.2. Компиляция, упаковка и запуск модуля monitor



Фраза «модули, указанные в командной строке» немного расплывчата. Теперь вы знаете, что это артефакты, которые можно найти в пути модуля.

Обратите внимание: я сказал *артефакты*, а не *модули*! Не только модульные JAR-файлы, но и обычные JAR, размещенные в пути модуля, превращаются в модули и становятся частью совокупности обозреваемых модулей. Это несколько удивительное поведение — часть истории миграции, и ее обсуждение здесь может помешать нашему исследованию пути модуля, поэтому позвольте мне отложить объяснение до раздела 8.3. Теперь я хочу упомянуть, что симметрично пути модуля, интерпретирующего каждый артефакт как модуль, путь класса обрабатывает все артефакты как простые JAR-файлы, независимо от того, содержат ли они дескриптор модуля.

> **Процессоры аннотаций**
>
> Если вы использовали процессоры аннотаций, то помещали их в путь класса вместе с артефактом приложения. Java 9 предлагает разделить их по задачам и применять `--class-path` или `--module-path` для JAR-файлов приложения и `--processor-path` или `--processor-module-path` для файлов JAR процессора. Для немодулированных JAR различие между путями приложения и процессора является факультативным: размещение обоих на пути к классам допустимо, но для модулей это обязательно; процессоры на пути модуля не будут использоваться.

Поскольку путь модуля используют несколько инструментов, особенно компилятор и виртуальная машина, имеет смысл взглянуть на концепцию в целом. Если не указано иное, то описанные механизмы работают одинаково во всех средах.

3.4.1. Регулирование модулей: анализ и верификация структуры приложения

Что происходит после вызова `javac` или `java` со связкой модулей в пути модуля? Именно тогда система модулей начинает проверять конфигурацию запуска (то есть модули и их объявленные зависимости) на надежность.

Процесс должен где-то начинаться, поэтому первым делом система модулей выбирает набор корневых модулей. Есть несколько способов сделать модуль корневым,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

и мы обсудим их все в свое время, но наиболее распространено указание исходного модуля. Для компилятора это либо компилируемый модуль (если декларация модуля находится среди исходных файлов), либо тот, который указан с параметром `--module` (если используется исходный путь модуля). В случае запуска виртуальной машины остается только вариант с `--module`.

Далее система модулей распределяет зависимости. Она проверяет декларации корневых модулей, чтобы увидеть, от каких других модулей они зависят, и пытается разрешить каждую зависимость с помощью наблюдаемого модуля. Затем продолжает делать то же самое с теми модулями и т. д. Это продолжается до тех пор, пока все переходные зависимости исходного модуля не будут распределены или конфигурация не будет определена как ненадежная.

> **Распределение сервисов и необязательных зависимостей**
>
> Два аспекта распределения модулей добавляют немного информации к обсуждаемому процессу:
>
> - сервисы (см. главу 10, в частности раздел 10.1);
> - необязательные зависимости (см. раздел 11.2, в частности подраздел 11.2.3).
>
> Я не буду вдаваться в подробности здесь, поскольку у нас нет необходимых условий, но хотел бы упомянуть их, поэтому теперь вы знаете: грядет нечто большее. Достаточно сказать, что это не аннулирует все предыдущее, а просто добавляет части к головоломке.

ВАЖНАЯ ИНФОРМАЦИЯ

Касаемо ненадежных настроек, в подразделе 3.2.2 рассматривалось все, что может пойти не так на данном этапе, и то, как система модулей реагирует на это. Следует добавить немаловажную деталь: если путь модуля состоит из нескольких записей (каталогов или отдельных JAR-файлов), то проверки на однозначность к ним не применяются! Каждая отдельная запись должна содержать модуль только один раз; но если несколько разных записей содержат один и тот же модуль, то выбирается первая (в порядке их нахождения в пути модуля) — она затеняет другие модули.

Самый простой способ показать, что модули могут дублироваться в разных папках, — выбрать проект, готовый к запуску и содержащий все модули в отдельной папке (скажем, `mods`). Затем нужно создать копию всей папки (скажем, `mods-copy`) и поместить обе в путь модуля:

```
$ java
 --module-path mods:mods-copy:libs
 --module monitor
```

Все модули появляются один раз в каждой папке, но приложение все равно запускается.

Теперь учтем, что инструменты сборки обычно создают путь модуля, который перечисляет каждую зависимость в отдельности. То есть пока инструмент сборки на-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ходится под контролем, например, во время компиляции и тестирования, проверки на однозначность не применяются ко всем зависимостям.

Я нахожу данный факт неудачным, поскольку он лишает нас возможности гарантировать надежную настройку. С другой стороны, у него есть преимущество: можно целенаправленно создавать теневые модули с версиями, которые вам по душе, после установки первой. Просто помните: в отличие от пути класса разные файлы JAR никогда не «смешиваются». Если система модулей выбирает один модуль как источник пакета, то будет искать все классы из него в этом JAR-файле и *никогда* не будет искать в других JAR (это относится к разделенным пакетам, обсуждаемым в подразделе 3.2.2 и разделе 7.2).

Далее, предположим, что все модули были распределены. Если ошибок не обнаружено, то система модулей гарантирует наличие каждого обязательного модуля. Вернее, то, что все модули с правильными именами есть в наличии.

На данном этапе никаких дополнительных проверок нет, поэтому если модуль зависит, например, от *com.google.common* (имя модуля для библиотеки Guava от Google) и найден пустой модуль с таким именем, то система модулей удовлетворится этим. Но пропущенные типы будут все еще вызывать проблемы в дальнейшем в виде ошибок во время компиляции или выполнения. В то время как пустые модули маловероятны, модуль с версией, отличной от ожидаемой, не содержащей пару типов, не кажется неправдоподобным. Тем не менее качественная настройка значительно уменьшит количество ошибок `NoClassDefFoundErrors`, возникающих во время выполнения.

3.4.2. Граф модуля: представление архитектуры приложения

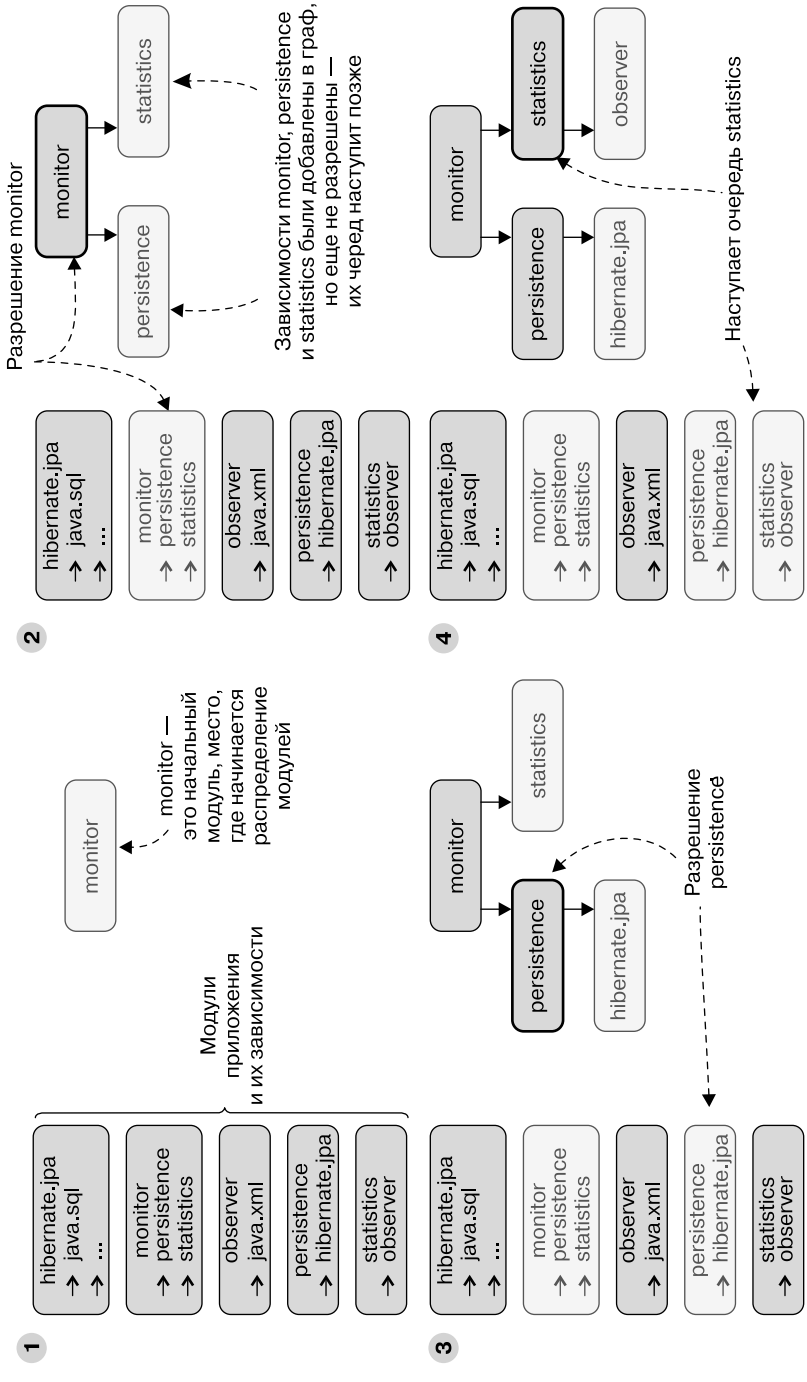
Один из первых заголовков книги — «Визуализация ПО в виде графов» (подраздел 1.1.1). В следующих рубриках объясняется, как разработчики и инструменты склонны рассматривать код в целом, но особенно зависимости между артефактами в виде графов. В оставшейся части главы 1 показано, что Java вместо этого рассматривает их в качестве простых контейнеров для типов, которые в результате превращаются в комок грязи, и что данное несоответствие является корнем нескольких серьезных проблем, поражающих экосистему.

Определение: граф модуля

В *графе модулей* модули (как узлы) связаны в соответствии с их зависимостями (с направленными ребрами). Ребра — основа для читабельности (описано в разделе 3.2). Граф строится во время распределения модуля и доступен во время выполнения через API рефлексии (обсуждается в подразделе 12.4.2).

На рис. 3.11 показано, как распределение модуля создает граф модуля для упрощенного приложения *ServiceMonitor*. Я думаю, не стоит возлагать всю работу на JPMS. Имея правильные параметры командной строки, можно добавить еще больше модулей и читать ребра графа, и мы рассмотрим это позже.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/



Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Рис. 3.11. Распределение модуля строит график модуля для упрощенного приложения ServiceMonitor. На каждом шаге разрешается один модуль, то есть он находится в совокупности обозреваемых модулей, и его зависимости добавляются в граф модулей. Шаг за шагом все переходные зависимости разрешаются, и в какой-то момент происходит переход от приложений к платформенным модулям

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

3.4.3. Добавление модулей в граф

Важно отметить, что модули, которые не попали в граф во время распределения, также не будут доступны позже во время компиляции или выполнения. Для случаев, когда весь код приложения находится в модулях, это часто не имеет значения. В конце концов, следуя правилам читабельности и доступности, даже будь такие модули доступными, их типы были бы недоступны, поскольку никто не читает данные модули. Но есть сценарии, использующие более сложные функции, где это может появиться в виде ошибки во время компиляции или выполнения либо даже в виде неправильного поведения приложения.

Различные варианты использования могут привести к тому, что модули не попадут в граф. Один из них — рефлексия. Она может применяться для того, чтобы код в одном модуле вызывался в другом, не имея явной зависимости от него. Но без этой зависимости зависимый модуль может не попасть в граф.

Предположим, существует некий альтернативный модуль статистики, *monitor.statistics.fancy*, который не может присутствовать в пути модуля для каждого развертывания сервиса. (Причина не имеет значения, но возьмем для примера лицензию, запрещающую использовать причудливый код во зло. Мы — злые вдохновители, иногда хотим сделать именно это.) Так что модуль может иногда присутствовать, а иногда нет, и, следовательно, никакой другой модуль не может его запрашивать, поскольку тогда приложение не запустится, если модуль будет отсутствовать.

Как приложению справиться с этим? Код, зависящий от библиотеки статистики, может использовать рефлексию для проверки наличия библиотеки и вызывать ее только в том случае, если она есть. Но согласно полученным только что знаниям, *это никогда не произойдет*! В силу необходимости *monitor.statistics.fancy* не требуется ни одному другому модулю и, следовательно, не попадет в граф, то есть никогда не будет вызван. Для этих и других сценариев, которые появляются на протяжении всей книги, система модулей предлагает решение.

Определение: --add-modules

Опция `--add-modules ${`*модули*`}`, доступная в `javac` и `java`, берет разделенный запятыми список имен модулей и определяет их как корневые за пределами исходного модуля. (Как сказано в подразделе 3.4.1, корневые модули образуют начальный набор модулей, из которых строится модульный граф путем распределения их зависимостей.) Это позволяет пользователям добавлять модули (и их зависимости) в граф, которые в противном случае не отображались бы, поскольку исходный модуль прямо или косвенно не зависит от них.

Опция `--add-modules` имеет три специальных значения: `ALL-DEFAULT`, `ALL-SYSTEM` и `ALL-MODULE-PATH`. Первые два работают только во время выполнения и используются для крайних случаев, которые не обсуждаются в данной книге. Последний может быть полезен, однако обратите внимание: с ним все модули на пути модуля становятся корневыми и, следовательно, трансформируются в модульный граф.

Если приложение *ServiceMonitor* имеет необязательную зависимость от *monitor.statistics.fancy*, то необходимо убедиться, что модуль отображается в графе для

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

тех развертываний, которые поставляются с ним. В таких случаях используйте `--add-modules monitor.statistics.fancy` с целью сделать его корневым модулем, в результате чего система модулей добавит его и его зависимости в граф:

```
$ java
 --module-path mods:libs
 --add-modules monitor.statistics.fancy
 --module monitor
```

Полученный граф модулей показан на рис. 3.12.

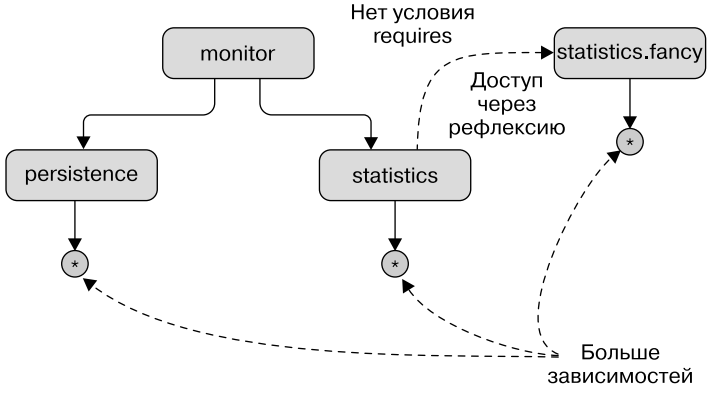


Рис. 3.12. Граф модулей для упрощенного приложения ServiceMonitor с рис. 3.10 с дополнительным корневым модулем monitor.statistics.fancy, определенным с помощью --add-modules. Ни модуль monitor, ни какая-либо из его зависимостей не зависят от него, поэтому он не сможет попасть в граф модулей без данной опции

Особенно важный пример использования `--add-modules` — модули JEE, которые, как объясняется в разделе 6.1, по умолчанию не распределяются при запуске приложения из пути класса. Модули можно добавлять в граф, отсюда естественно возникает вопрос, можно ли также удалить их. Ответ «да», например: опция `--limit-modules` подходит для этого, а в подразделе 5.3.4 показано, как она работает.

К сожалению, нельзя дать системе модулей знать, что конкретная зависимость не будет заполнена, с вашего позволения. Так можно было бы исключить ненужные (переходные) зависимости. Судя по количеству исключений, которые я вижу в типичных Maven POM, это распространено, но, увы, строгость системы модулей этого не позволяет.

3.4.4. Добавление ребер в граф

Когда модуль добавлен явно, он находится в графе сам по себе, без каких-либо входящих ребер читабельности. Если доступ к нему полностью рефлексивен, то это нормально, поскольку API рефлексии неявно добавляет ребро читабельности. Но для обычного доступа, например при импорте типа из него, правила доступности требуют читабельности.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Определение: --add-reads

Опция времени компиляции и выполнения `--add-reads ${`*`модуль`*`}=${`*`цели`*`}` добавляет ребра читаемости из `${`*`модуль`*`}` во все модули в разделенный запятыми список `${`*`цели`*`}`. Это позволяет `${`*`модуль`*`}` получать доступ ко всем публичным типам в пакетах, экспортируемых этими модулями, даже если в них нет директивы `requires`, в которых они упоминаются. Если `${`*`цели`*`}` включает в себя `ALL-UNNAMED`, то `${`*`модуль`*`}` может прочитать содержимое пути класса (это немного сложно; подробности см. в разделе 8.2).

Вернувшись к *monitor.statistics.fancy*, можно добавить `add-reads`, что позволит *monitor.statistics* прочитать его:

```
$ java
 --module-path mods:libs
 --add-modules monitor.statistics.fancy
 --add-reads monitor.statistics=monitor.statistics.fancy
 --module monitor
```

Полученный граф модулей такой же, как на рис. 3.12, за исключением того, что пунктирная линия теперь заменена на правильное ребро чтения. В конце подраздела 8.3.2 есть пример, когда `--add-reads… =ALL-UNNAMED` спасает положение.

3.4.5. Доступность — постоянная цель

Как только система модулей распределит все зависимости, построит граф и установит читабельность между модулями, она останется активной, чтобы проверить соблюдение правил доступности, определенных в разделе 3.3. При их нарушении возникают ошибки во время компиляции или выполнения, как показано в подразделе 3.3.3. Если вы столкнулись с проблемой в системе модулей и из сообщения об ошибке не можете понять, что пошло не так, обратитесь к разделу 5.3 за советом, как отладить ситуацию.

Если вы хотите узнать больше о создании и запуске модульных приложений, таких как собственные «целинные» проекты, то главы 4 и 5 подробно расскажут об этом. Кроме того, в главах 6 и 7 вы можете проверить, как система модулей влияет на существующий проект. Вы также хорошо подготовлены к тому, чтобы углубиться и проверить расширенные функции.

ПРИМЕЧАНИЕ

Вы достигли рубежа! Теперь вы понимаете, как определяются модули, какие механизмы работают с этим определением и как влияют — в целом, как Java работает с модулями.

Резюме

- Модули бывают двух видов.
  - Поставляемые с Java Runtime — *платформенные модули*. Они объединены в файл `modules` в каталоге `libs` среды выполнения. JDK также хранит их

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

в необработанном виде как файлы JMOD в каталоге `jmods`. Только *java.base, базовый модуль*, явно известен системе модулей.

- Разработчики библиотек, фреймворков и приложений создают *модульные JAR-файлы*, которые являются простыми JAR-файлами, содержащими *дескриптор модуля* `module-info.class`. Они называются *прикладными модулями*, один из которых содержит метод `main`, являющийся *начальным модулем*.

- Дескриптор модуля составлен из *объявления модуля module-info.java,* который могут редактировать разработчики (и инструменты). Он лежит в основе работы с системой модулей и определяет свойства модуля:
  - его имя, которое должно быть глобально уникальным по обратной схеме доменных имен;
  - его зависимости, указанные с помощью директивы `requires`, чтобы ссылаться на другие модули по имени;
  - его API, который определяется путем экспорта выбранных пакетов с директивами `exports`.
- Декларации зависимостей и *ребра читабельности*, которые создает из них система модулей, — основа для *надежной настройки*. Это достигается за счет того, что среди прочего все модули появляются ровно один раз и между ними не существует цикличных зависимостей. Это позволяет раньше выявлять проблемы, связанные с повреждением или сбоем приложения.
- Ребра читабельности и экспорт пакетов в совокупности являются основой для *надежной инкапсуляции*. Здесь система модулей обеспечивает доступность только открытых типов в экспортируемых пакетах и лишь для модулей, которые читают экспортирующий. Это предотвращает случайные связи с переходными зависимостями и позволяет убедиться, что внешний код не может зависеть от типов, которые вы разработали в качестве внутренних для модуля.
- Ограничения доступности применимы и к рефлексии! Это требует немного больше взаимодействия с основанными на рефлексии средами, такими как Spring, Guice или Hibernate.

Путь модуля (опция `--module-path` или `-p`) состоит из файлов или каталогов и делает JAR-файлы доступными для системы модулей, которая будет представлять их как модули. Используйте путь модуля вместо пути класса, чтобы компилятор или JVM знали об артефактах вашего проекта.

Модули приложения, указанные в пути модуля, и платформенные модули, появляющиеся во время выполнения, составляют *совокупность обозреваемых модулей*. Во время распределения в совокупности выполняется поиск модулей, начиная с корневых, поэтому все необходимые модули должны либо находиться в пути модуля, либо вызываться во время выполнения.

Разрешение модуля проверяет, является ли конфигурация надежной (присутствуют все зависимости, нет ли неоднозначностей и т. д., как было описано в разделе 3.2), и приводит к графу модулей — близкому представлению в системе модулей того, как мы видим зависимости артефактов. Во время выполнения доступны только модули, которые входят в граф.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

4 Построение модулей от исходного кода до JAR

В этой главе

- Структуры каталогов проекта.
- Компиляция исходников от отдельного модуля до файлов классов.
- Компиляция нескольких модулей одновременно.
- Упаковка файлов классов в модульный JAR-файл.

Умение определять модули, как описано в главе 3, — хороший навык, но зачем он нужен, если вы не знаете способа превратить эти исходные файлы в модульные артефакты (JAR), которые можно отправлять и запускать? В этой главе рассматривается создание модулей, начиная с организации исходного кода и заканчивая их компиляцией в файлы классов и, в итоге, упаковкой их в модульные JAR-файлы, которые можно распределять и выполнять. Глава 5 посвящена запуску и отладке модульных приложений.

Периодически мы будем рассматривать команды `javac` и `jar`, доступные в командной строке. Можно удивиться: разве IDE и другие инструменты уже не используют их самостоятельно? Вероятно, да, но, даже если оставить в стороне аргументы о том, что всегда полезно знать, как данные инструменты творят свою магию, есть более важная причина изучить эти команды: они представляют собой самый прямой путь в сердце системы модулей. Мы будем использовать их для исследования ее функций изнутри и снаружи, и после окончания вы сможете применить любой инструмент, который дает доступ к этим функциям.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Первое, что мы рассмотрим в данной главе, — как файлы проекта должны быть организованы на диске (раздел 4.1). Это может показаться банальным, но стоит изучить все, не пропуская ни одного шага. Имея размеченные источники и модули, объявленные согласно описанию в главе 3, мы перейдем к их компиляции. Она может проводиться по одному модулю за раз (см. раздел 4.2) или по несколько одновременно (см. раздел 4.3). В последнем разделе обсудим, как упаковать файлы классов в модульные JAR. Чтобы рассмотреть некоторые реальные сценарии сборки, изучите ветку `master` из *ServiceMonitor*.

К концу этой главы вы сможете организовать, скомпилировать и упаковать исходный код и декларации модулей. Получившиеся модульные JAR-файлы готовы к развертыванию или отправке любому, кто применяет Java 9 или более позднюю версию и готов использовать все преимущества модулей.

4.1. Структурирование проекта с помощью каталогов

Реальный проект состоит из множества файлов разных типов. Очевидно, что исходные файлы наиболее важны, однако тем не менее они представляют собой лишь один из многих типов: другие — это тесты, ресурсы, сценарии сборки или описания проектов, документация, информация об управлении версиями и др. В любом проекте следует выбрать структуру каталогов для организации этих файлов и важно убедиться, что она не конфликтует с характеристиками системы модулей.

Если вы следили за разработкой системы модулей в Project Jigsaw и изучали официальное краткое руководство или ранние учебные пособия, то могли заметить, что они используют определенную структуру каталогов. Рассмотрим рекомендацию, чтобы проверить, может ли она стать новым соглашением, и сопоставить ее со значением по умолчанию, которое неявно распознается такими инструментами, как Maven и Gradle.

4.1.1. Новое предложение — новое соглашение?

В ранних публикациях, посвященных системе модулей, каталог проекта часто содержит каталог `src`, в котором каждый модуль, принадлежащий проекту, имеет собственный подкаталог, содержащий исходные файлы проекта. Если проекту нужны не только исходники, то предложение состоит в том, чтобы организовать эти задачи в параллельных деревьях с такими папками, как `test` и `build` рядом с `src`. Результат — иерархия `concern/module`, показанная на рис. 4.1.

Важно понять, что это за структура единого `src`: структура конкретного проекта (JDK) и предложение, использованное во введении. Из-за тенденции разбивать файлы одного модуля между параллельными деревьями я бы не советовал применять ее для чего-либо, кроме самых маленьких проектов или проектов, тщательный анализ которых показывает, что эта структура предпочтительнее. В противном случае лучше задействовать установленную по умолчанию структуру, которую мы обсудим далее.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

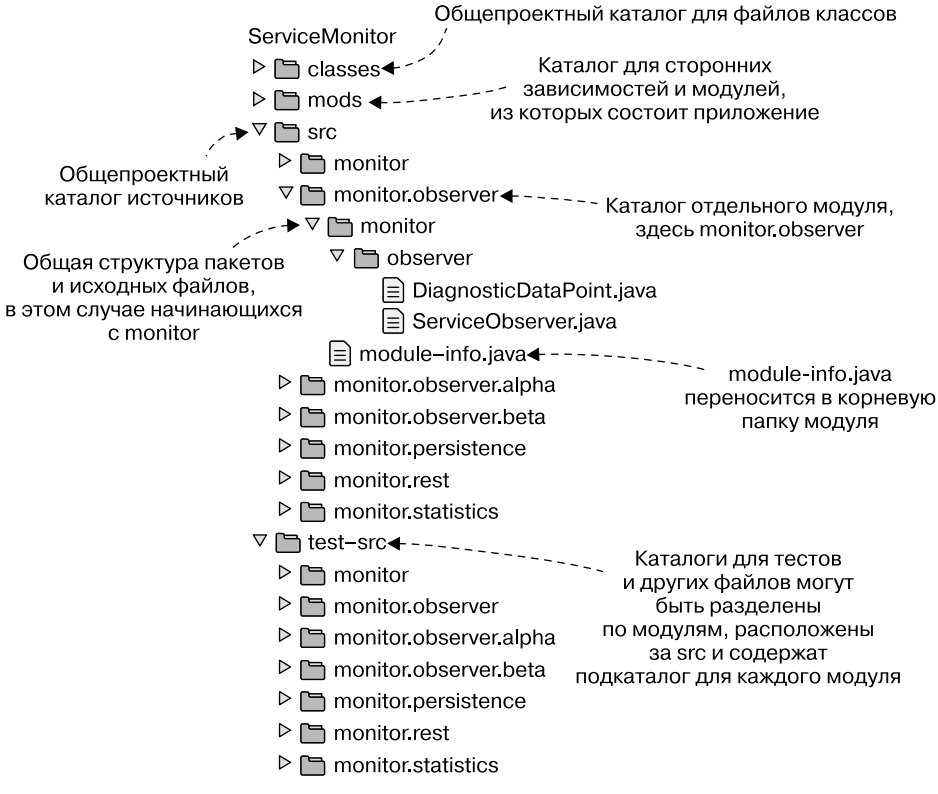


Рис. 4.1. В структуре находятся каталоги верхнего уровня: classes, mods, src и testsrc. Источники отдельных модулей содержатся в каталогах ниже src или test-src, которые называются так же, как и модуль

4.1.2. Созданная структура каталогов

Большинство проектов, которые состоят из нескольких подпроектов (которые мы теперь называем *модулями*), предпочитают отдельные корневые каталоги, содержащие все источники, тесты, ресурсы и все остальное, упомянутое ранее. Они используют иерархию `concern/module`, и это именно то, что предлагают установленные структуры проекта.

Структура каталогов по умолчанию, неявно распознаваемая такими инструментами, как Maven и Gradle, реализует эту иерархию (рис. 4.2). Прежде всего, структура по умолчанию дает каждому модулю собственное дерево каталогов. В данном дереве каталог `src` содержит эксплуатационный код и ресурсы (в `main/java` и `main/resources` соответственно), а также тестовый код и ресурсы (в `test/java` и `test/resources` соответственно).

Нет необходимости структурировать все проекты подобным образом. Если оставить в стороне дополнительную работу по настройке инструментов сборки для каталогов, отклоняющихся от выбранной схемы, и конкретный случай многомодульной

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

компиляции (описанный в разделе 4.3.), то все структуры одинаково действительны и должны быть выбраны исходя из их преимуществ для данного проекта.

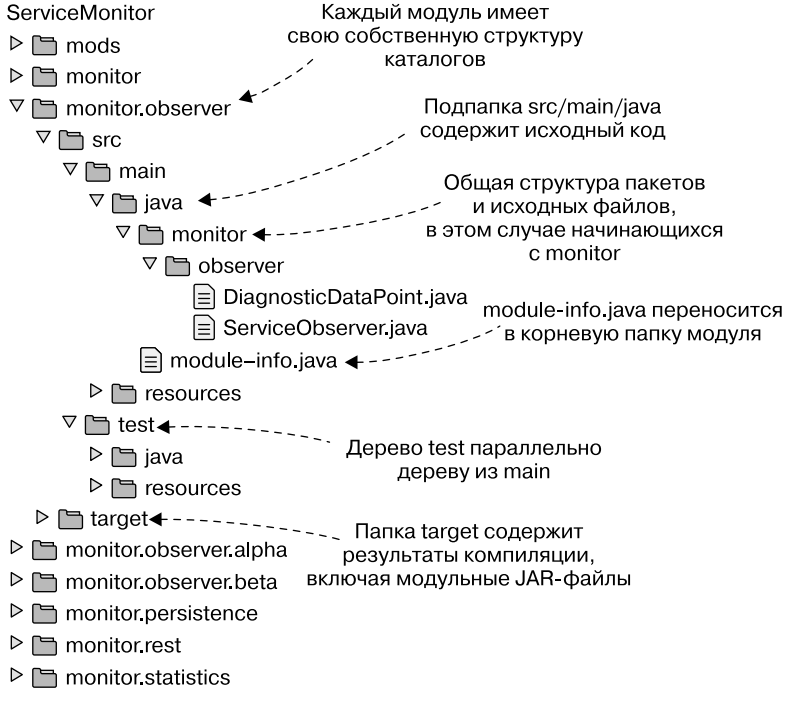


Рис. 4.2. В данной структуре есть каталог верхнего уровня для каждого модуля. Модули могут организовывать собственные файлы в соответствии с их потребностями. Здесь monitor.observer применяет общую структуру каталогов, используемую в проектах Maven и Gradle

Учитывая все вышесказанное, в примерах, приведенных в этой книге, используется подобная структура по умолчанию с одним исключением: применение командной строки становится менее громоздким, если все модульные JAR-файлы оказываются в одном и том же каталоге, поэтому дерево приложения *ServiceMonitor* содержит папку `mods` верхнего уровня, в которой находятся созданные модули.

4.1.3. Место для деклараций модулей

При любом способе структурирования исходных файлов декларации модулей должны называться `module-info.java`. В противном случае компилятор выдает ошибку, подобную указанной ниже, во время попытки компиляции `monitor-observer-info.java`:

```
> monitor.observer/src/main/java/monitor-observer-info.java:1:
>     error: module declarations should be in a file named module-info.java
> module monitor.observer {
> ^
> 1 error
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Хоть это и не строго необходимо, декларация должна находиться в корневом каталоге исходного кода. В противном случае использование пути к исходному модулю, как описано в подразделе 4.3.2, не будет работать должным образом, поскольку система модулей не сможет найти дескриптор. Вследствие этого она не распознает модуль, что приводит к таким ошибкам, как «модуль не найден».

Чтобы в этом убедиться, переместите дескриптор *monitor.observer* в другой каталог и скомпилируйте *monitor*. Как видите, это приводит к ошибке, что модуль *monitor.observer*, который требуется для *monitor*, не может быть найден:

```
> ./monitor/src/main/java/module-info.java:2:
>     error: module not found: monitor.observer
>         requires monitor.observer;
>                  ^
> 1 error
```

4.2. Компиляция отдельного модуля

Как только файлы проекта размещены в структуре каталогов, написано немного кода и созданы декларации модулей, наступает пора компилировать исходные файлы. Но что получится в результате — коллекция типов или новенький блестящий модуль? Поскольку первое не менялось, мы сосредоточимся на последнем, прежде чем исследовать, как компилятор распознает оба случая.

4.2.1. Компиляция модульного кода

Данный подраздел посвящен компиляции единственного модуля в мире, зависимости которого уже являются модульными. Можно скомпилировать модуль, только если декларация `module-info.java` находится среди исходных файлов, поэтому предположим, что это так.

Помимо работы с путем модуля и проверки читабельности и доступности, еще одним дополнением к компилятору является его способность обрабатывать декларации модулей. Результат компиляции декларации модуля — дескриптор модуля, файл `module-info.class`. Как и другие файлы `.class`, он содержит байт-код и может анализироваться и обрабатываться такими инструментами, как ASM и Apache's Byte Code Engineering Library (BCEL).

Помимо использования пути модуля вместо пути класса, компиляция работает точно так же, как и до Java 9. Компилятор скомпилирует все заданные файлы и создаст структуру каталогов, соответствующую иерархии пакетов в выходном каталоге, указанном с помощью `-d`.

На рис. 4.3 показано, как устроен модуль *monitor.observer*, который использует структуру каталогов по умолчанию. Для его компиляции необходимо организовать вызов `javac`, что похоже на действия, выполняемые вами до Java 9:

- опция `--module-path` указывает компилятору на каталог, в котором содержатся необходимые модули приложения;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- опции `-d` определяют целевой каталог для компиляции; они работают так же, как и до Java 9;
- `find` используется для поиска всех исходных файлов в `monitor.observer/src/main/java/`, включая `module-info.java` (представлен как `${`*`исходные_файлы`*`}`).

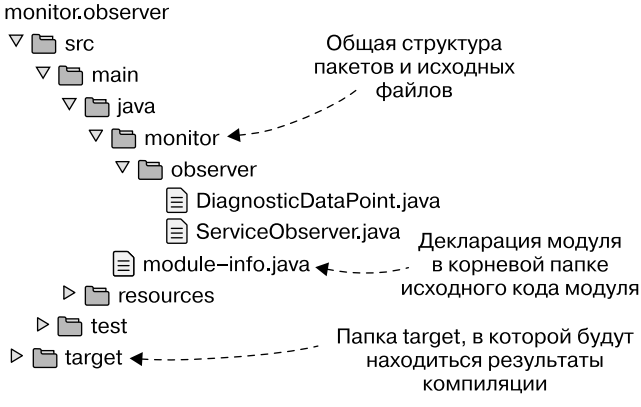


Рис. 4.3. Структура каталогов модуля monitor.observer с развернутым каталогом src

Собрав все вместе, введите следующую команду в корневом каталоге приложения *ServiceMonitor* (то есть содержащем *monitor.observer*):

```
$ javac
    --module-path mods
    -d monitor.observer/target/classes
    ${исходные_файлы}
```

На рис. 4.4 `src` свернут, в то время как нас интересует `target/classes`.

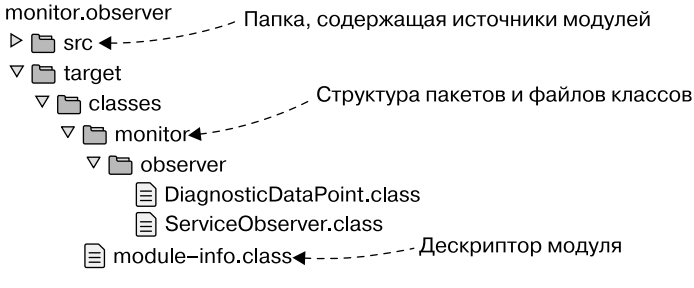


Рис. 4.4. Структура каталогов модуля monitor.observer с развернутым каталогом target

4.2.2. Модульное или немодульное?

Система модулей платформы Java построена с целью создания и в конечном итоге запуска модулей, но это ни в коем случае не является обязательным. Все еще можно создать простые JAR-файлы, а потому возникает вопрос о том, как различаются эти два случая. Как компилятор узнает, что создавать: модуль или группу типов?

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

Как обсуждалось в подразделе 3.1.2, модульный JAR — не что иное, как простой JAR с дескриптором модуля module-info.class, который скомпилирован из декларации модуля module-info.java. По этой причине компилятор принимает во внимание наличие или отсутствие module-info.java в списке источников во время компиляции, чтобы определить, модуль это или нет. Вот почему не существует опции компилятора --create-module или чего-то подобного.

В чем разница между компиляцией модуля и компиляцией только типов? Все сводится к читабельности, как описано в разделе 3.2. Если код, который включает в себя объявление модуля, скомпилирован, то:

- должен содержать требование, чтобы его зависимости имели доступ к типам, которые экспортируются этими зависимостями;
- необходимые зависимости должны быть представлены.

С другой стороны, если компилируется немодульный код, то зависимости не выражаются из-за отсутствия объявления модуля. В этом случае система модулей позволяет компилируемому коду читать все модули и все остальное, что она находит в пути классов. Раздел 8.2 подробно описывает этот *режим пути классов*.

В отличие от читабельности правила доступности, описанные в разделе 3.3, применяются в обоих случаях. Независимо от того, как код скомпилирован: в виде модуля или группы исходников, он связан с правилами доступа к типам в других модулях. Это особенно актуально в отношении внутренних классов JDK, будь то открытые классы в неэкспортированных пакетах или непубличные классы, поскольку они недоступны независимо от того, как компилируется код. На рис. 4.5 показана разница между читабельностью и доступностью.

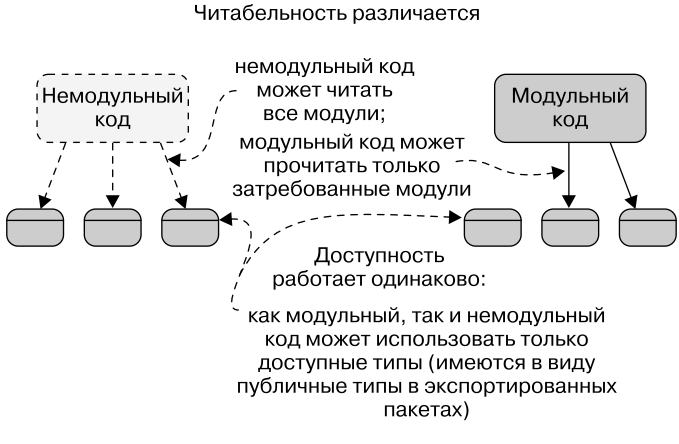


Рис. 4.5. Сравнение компиляции немодульного кода *(слева)* с модульным кодом *(справа)*. Правила читабельности немного различаются, тогда как правила доступности идентичны

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Заметка об ошибках компилятора

Рассмотрим приложение *ServiceMonitor* в качестве примера. Его подпроект *monitor* содержит исходные файлы `Main.java`, `Monitor.java` и `module-info.java`.

Если включить декларацию модуля в список файлов, `javac` соберется скомпилировать именно модуль и проверит, все ли зависимости от модулей приложения и платформенных модулей объявлены в дескрипторе. Если пропустить ее, то компилятор вернется к распознаванию зависимостей между типами, как показано на рис. 3.1 (см. выше).

Но независимо от того, скомпилирован ли *monitor* как модуль, если он использует типы, которые модули JDK или другие модули приложения не делают доступными, то результат будет одинаковым: ошибка компиляции.

Компиляция модуля, очевидно, требует устранения большего количества препятствий, чем компиляция только типов. Зачем тогда она нужна? Я снова возвращаюсь к сравнению с написанием кода на статически типизированном языке. Как разработчики Java, мы обычно считаем, что статическая типизация стоит дополнительных первоначальных затрат, поскольку взамен получаем быстрые и надежные проверки согласованности. Они предотвращают не все ошибки, но многие из них.

То же самое применимо и здесь: использование системы модулей для компиляции модулей требует больше усилий, чем создание простых JAR-файлов, но взамен мы получаем проверки, уменьшающие вероятность ошибок времени выполнения. Мы обмениваем усилия на этапе компиляции на безопасность времени выполнения — сделка, которую я могу заключить в любой день недели.

4.3. Компиляция нескольких модулей

Компиляция одного модуля, как было описано выше, проста, а компиляция всех семи модулей *ServiceMonitor* — даже еще проще. Однако нужно ли компилировать модули один за другим? Или, если посмотреть на это иначе, есть ли причина не делать так? Ответ на последний вопрос — да, несколько факторов могут сделать предпочтительной компиляцию нескольких модулей одновременно.

- *Усилие* — компиляция одного модуля проста, однако усилия, необходимые для нескольких модулей, быстро накапливаются. И безусловно, повторение одной и той же команды снова и снова с небольшими изменениями кажется излишним. Скорее всего, вы редко будете делать это вручную, если только не экспериментируете с Java 9. Но следует учитывать и позицию разработчиков, пользующихся вашими инструментами.
- *Производительность* — компиляция дескриптора одного модуля в моей системе занимает около полусекунды, а компиляция всех модулей приложения *ServiceMonitor* — около четырех. Это немного, учитывая, что задействовано менее 20 исходных файлов, а полная сборка гораздо более крупных проектов занимает

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

меньше времени. Разумеется, я плачу за запуск компилятора семь раз (за семь модулей).

- *Слабые циклические зависимости* — несмотря на то что система модулей запрещает циклические зависимости с помощью директив `require`, существуют другие способы заставить модули ссылаться друг на друга (просто поверьте сейчас), которые считаются приемлемыми. Хоть зависимости и циклические, их можно считать слабыми, поскольку при отсутствии одной из них вы получите только предупреждение. Тем не менее компиляция без предупреждений стоит некоторых усилий, и для этого оба модуля должны быть собраны вместе.

ВАЖНАЯ ИНФОРМАЦИЯ

Поскольку у нас уже набралось несколько причин для компиляции нескольких модулей одновременно, хорошо, что компилятор умеет делать это!

4.3.1. Безыскусный подход

Как работает компиляция нескольких модулей одновременно? Можно ли перечислить исходные файлы из нескольких модулей и заставить компилятор выяснить это? Нет:

```
$ javac
    --module-path mods:libs
    -d classes
    monitor/src/main/java/module-info.java
    monitor.rest/src/main/java/module-info.java

> monitor.rest/src/main/java/module-info.java:1:
>     error: too many module declarations found
> module monitor.rest {
> ^
> 1 error
```

Очевидно, что компилятор предпочитает работать с одним модулем в одно время. Это имеет смысл, поскольку, как обсуждалось ранее, он обеспечивает читабельность и доступность на основе четко определенных границ модуля. Откуда они берутся, с источниками из разных модулей, смешанными в списке файлов для компиляции? Каким-то образом компилятор должен знать, где заканчивается один модуль и начинается следующий.

4.3.2. Путь к исходнику модуля: информирование компилятора о структуре проекта

Выход из данного режима по умолчанию для одного модуля — параметр командной строки, информирующий компилятор о структуре каталогов проекта. Компилятор поддерживает *многомодульную компиляцию*, в которой может создавать несколько

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

модулей одновременно. Параметр командной строки `--module-source-path ${`*путь*`}` используется с целью включить этот режим и указать структуру каталогов, содержащую модули. Все остальные параметры компилятора работают как обычно.

Описанное звучит довольно просто, однако нужно учесть некоторые важные детали. Прежде чем сделать это, рассмотрим простой пример.

На минуту предположим, что приложение *ServiceMonitor* использовало структуру единого `src`, определенную в разделе 4.1.1, со всеми исходными каталогами модулей под `src` (рис. 4.6). Затем можно было применить `--module-source-path src` с целью указать компилятору на папку `src`, в которой содержатся все исходники модулей, и сказать ему, чтобы он компилировал все, что обнаружил, одновременно.

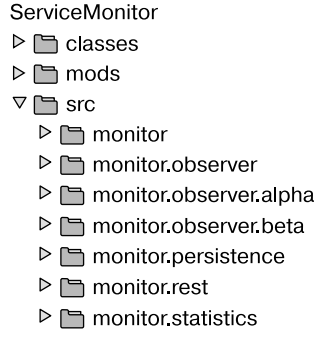


Рис. 4.6. Путь к исходнику модуля проще всего использовать, если в проекте имеется один каталог src с корневым исходным каталогом каждого модуля под ним

Как и в случае сборки с одним модулем, путь модуля используется с целью указать компилятору на каталог, в котором содержатся необходимые модули приложения — в данном случае это внешние зависимости, поскольку все модули *ServiceMonitor* компилируются в настоящее время. Опция `-d` работает так же, как и при сборке одного модуля, и по-прежнему нужно перечислять все исходные файлы в `src`, включая все декларации модулей.

Собрав все это вместе, получим команду:

```
$ javac
    --module-path mods:libs
    --module-source-path src
    -d classes
    ${исходные_файлы}
```

Просмотр `classes` показывает каталоги модулей, каждый из которых содержит файлы классов данного модуля, включая дескриптор модуля. Прекрасно.

Но это не всегда так просто. Как все вышеописанное применимо к проекту, в котором не используется структура единого `src`? Вот тут и появляется изящная деталь исходного пути модуля.

4.3.3. Звездочка как токен для имени модуля

Исходный путь модуля может содержать звездочку (*). Как правило, это интерпретируется как подстановочный знак, который в путях обычно означает «что-нибудь в каталоге вплоть до звездочки». Однако в данном случае все не так. Вместо этого звездочка действует как маркер, указывающий, где на пути появляются имена модулей. Остальная часть пути после звездочки должна указывать на каталог, содержащий пакеты модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Таким образом, компилятор может сопоставить путь к исходному файлу с путем к модулю и определить, к какому модулю принадлежит исходный файл. Чтобы это работало, каждый исходный файл должен соответствовать пути модуля.

Это может показаться сложным, но все станет ясно на примере. Вернемся к приложению *ServiceMonitor*, структурированному в подразделе 4.1.2, где каждый модуль имеет общие каталоги `src/main/java`, содержащие исходные файлы. Начиная с верхнего уровня проекта, это относительные пути к некоторым источникам:

- `monitor/src/main/java/monitor/Monitor.java`;
- `monitor/src/main/java/monitor/Main.java`;
- `monitor/src/main/java/module-info.java`;
- `monitor.rest/src/main/java/monitor/rest/MonitorServer.java`;
- `monitor.rest/src/main/java/module-info.java`;
- `monitor.persistence/src/main/java/monitor/persistence/StatisticsRepository.java`;
- `monitor.persistence/src/main/java/module-info.java`.

Это делает общую структуру довольно очевидной: все пути следуют схеме `${`*`модули`*`}/src/main/java/${`*`пакеты`*`}/${`*`исходные файлы`*`}`.

Оглядываясь назад на то, как должен использоваться исходный путь модуля, вы можете вспомнить, что необходимо заменить `${`*`модули`*`}` на `*` и нужно пропустить каталоги пакетов, оставив `*/src/main/java`. К сожалению, это еще не сработает, поскольку компилятор не принимает звездочку в качестве первого символа — необходимо дополнить его `./`. Теперь многомодульная компиляция работает как по волшебству:

```
$ javac
    --module-path mods:libs
    --module-source-path "./*/src/main/java"
    -d classes
    ${исходные_файлы}
```

Как и прежде, все файлы классов попадают в специфичные для модуля подкаталоги `classes`. Зная, что звездочка — токен для имени модуля, можно суммировать эти пути как `-d classes/*`. К сожалению, опция `-d` не понимает токен и потому нельзя использовать его в целях создания выходных путей, таких как `./*/target/classes`. Ужасно.

Вы можете задаться вопросом, как звездочка относится к использованию `--module-source-path src` в первом примере. В конце концов, вы не указали, где будут появляться имена модулей, а компилятор смог их вывести. На первый взгляд то, что может выглядеть как несоответствие, — попытка сделать простое применение еще более простым.

Если исходный путь модуля не содержит звездочку, то компилятор автоматически добавит ее как окончательный элемент пути. Таким образом, вы фактически

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

указали `src/*` в качестве исходного пути модуля, который соответствует структуре каталогов в этом примере.

Возможность компилировать несколько модулей, если все они используют одну и ту же структуру каталогов, должна охватывать большинство случаев применения. Для более сложных настроек нам нужен другой метод.

4.3.4. Несколько путей к исходникам модулей

Возможно, одного пути к исходному модулю недостаточно. Вероятно, у разных модулей разные структуры каталогов или некоторые модули имеют источники в нескольких каталогах. В таких случаях можно указать несколько путей к модульным исходникам, чтобы убедиться в соответствии каждого исходного файла выбранному пути.

JDK, будучи сложным проектом, имеет нетривиальную структуру каталогов. На рис. 4.7 показан лишь небольшой фрагмент кода — на всех уровнях существует еще больше каталогов.

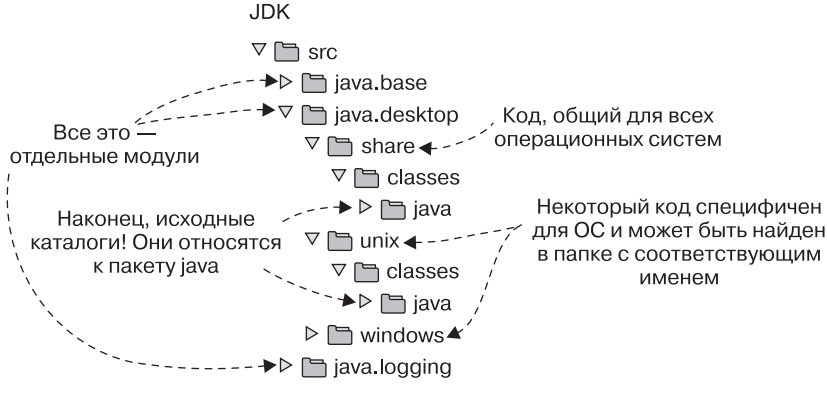


Рис. 4.7. Урезанный просмотр исходных каталогов JDK. Обратите внимание, как разделяются каталоги модулей ниже src. Ниже приведены каталоги classes, которые являются корневыми для исходных файлов

Если вы находитесь в каталоге `jdk` и хотите провести сборку для UNIX, то как будет выглядеть исходный путь модуля, который охватывает все модули и правильные исходные папки? Путь к источникам UNIX — `src/java.desktop/unix/classes` или в более общем смысле, `src/${`*модуль*`}/unix/classes`. Аналогично для общих источников это `src/${`*модуль*`}/share/classes`. Соединив их, получаем:

```
--module-source-path "src/*/unix/classes":"src/*/share/classes"
```

Чтобы уменьшить избыточность, для исходного пути модуля определим альтернативные пути с помощью `{dir1,dir2}`. Можно объединить разные пути, если

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

они различаются только названием отдельных элементов. Данная альтернатива позволяет унифицировать пути к источнику в `share` и `unix` следующим образом:

```
--module-source-path "src /*/{share, unix}/classes"
```

4.3.5. Настройка начального модуля

Когда все настроено для многомодульной компиляции, открывается еще одна возможность: компилировать отдельный модуль и его зависимости, просто указав его название. Зачем это нужно? Затем, чтобы больше не указывать исходные файлы для компиляции явно!

Если указан путь к исходному модулю, то опция `--module` позволяет скомпилировать отдельный модуль и его переходные зависимости *без* явного перечисления исходных файлов. Путь к модулю используется для определения того, какие исходные файлы принадлежат указанному модулю, а зависимости разрешаются на основе его объявления.

Компиляция *monitor.rest* и его зависимостей теперь она упростилась. Как и раньше, используйте `--module -path mods:libs`, чтобы указать, где искать зависимости, и `-d classes`, чтобы определить выходную папку. С помощью `--module-source-path "./*/src/main/java"` вы сообщаете компилятору о структуре каталогов вашего проекта и с помощью `--module monitor.rest` даете команду начать компиляцию *monitor.rest*:

```
$ javac
    --module-path mods:libs
    --module-source-path "./*/src/main/java"
    -d classes
    --module monitor.rest
```

Если раньше папка `classes` была пустой, то теперь содержит файлы классов для `monitor.rest` (указанный модуль), `monitor.statistics` (прямая зависимость) и `monitor.observer` (переходная зависимость).

В листингах 2.3, 2.4 и 2.5 шаг за шагом показана компиляция приложения *ServiceMonitor*. Вооружившись знанием о способе использования многомодульной компиляции, это можно сделать так же легко, как в примере ниже:

```
$ javac
    --module-path mods:libs
    --module-source-path "./*/src/main/java"
    -d classes
    --module monitor
```

Поскольку начальный модуль *monitor* зависит от всех других модулей, все они собраны. В отличие от пошагового подхода файлы классов расположены не в `*/target/classes`, а в `classes/*` (используем * в качестве токена для имени модуля).

Помимо упрощения чтения команды, комбинация `--modulesource-path` и `--module` также работает на более высоком уровне абстракции. В отличие от перечисления отдельных исходных файлов в ней четко указана цель компиляции определенного модуля. Мне это нравится.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Однако есть два недостатка.

- Скомпилированные файлы классов не могут быть перераспределены в более глубокие структуры каталогов, и вместо этого все они попадают в один и тот же каталог (в недавних примерах `classes`). Если последующие стадии процесса сборки зависят от точного местоположения этих файлов, то необходимо будет выполнить подготовительные шаги, что может лишить вас преимуществ использования пути к исходнику модуля.
- Если компиляция начинается с `--module` (в отличие от перечисления всех исходных файлов модуля), то компилятор применяет оптимизации, которые могут привести к неожиданным результатам. Одна из них — обнаружение неиспользуемого кода: классы, на которые исходный модуль не ссылается переходным способом, не компилируются, и даже целые модули могут отсутствовать в выходных данных, если были отделены с помощью сервисов (см. главу 10).

4.3.6. Стоит ли оно того?

Окупается ли многомодульная компиляция? Я перечислял три причины, чтобы мотивировать вас к ее использованию, поэтому имеет смысл вернуться к ним.

- *Усилия* — вслед за пониманием, как должен создаваться исходный путь модуля, вам потребуется значительно меньше усилий для компиляции нескольких модулей. Упростится и сборка конкретного модуля с его зависимостями. В то же время инструменты сборки обычно компилируют проекты один за другим и их одновременная настройка может усложнить задачу, особенно если необходимо предпринять дальнейшие шаги для распространения файлов классов в каталоги, специфичные для модулей.
- *Производительность* — при многомодульной компиляции приложение *ServiceMonitor* собирается менее чем за секунду, что в четыре раза быстрее, чем сборка семи модулей шаг за шагом. Но это довольно экстремальный случай, поскольку каждый модуль содержит только два или три класса. Условно говоря, запуск компилятора семь раз требует много работы; но в абсолютном выражении время, потраченное на это, сводится к трем секундам. Учитывая время сборки любого проекта приличного размера, едва ли стоит сбрасывать пару секунд ради усложнения сборки.
- *Слабые циклические зависимости* — если в сборке не должно быть предупреждений, то нет возможности обойтись без многомодульной компиляции.

Многомодульная компиляция необязательна, и ее преимущества не настолько существенны, чтобы рекомендовать ее в качестве стандартной практики. В частности, если ваши инструменты не поддерживают ее беспроблемно, то ее настройка может не стоить затраченных усилий. Это классическая ситуация «все зависит от». Я просто должен упомянуть, что мне нравится работать на более высоком уровне абстракции, используя модули вместо просто типов.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

4.4. Параметры компилятора

С системой модулей поставляется множество новых параметров командной строки, которые описаны в этой книге. Чтобы вам было проще их найти, в табл. 4.1 перечислены все относящиеся к компилятору. Для просмотра официальной документации по компилятору зайдите на docs.oracle.com/javase/9/tools/javac.htm.

Таблица 4.1. Таблица всех параметров компилятора, связанного с модулями, в алфавитном порядке (команда javac). Описания основаны на документации, а ссылки указывают на разделы в этой книге, в которых подробно объясняется, как использовать параметры

Параметр	Описание	Ссылка
--add-exports	Позволяет модулю экспортировать дополнительные пакеты	11.3.4
--add-modules	Объявляет корневые модули в дополнение к начальному	3.4.3
--add-reads	Добавляет ребра чтения между модулями	3.4.4
--limit-modules	Ограничивает совокупность обозреваемых модулей	5.3.5
--module, -m	Определяет начальный модуль	4.3.5
--module-path, -p	Указывает, где находятся модули приложения	3.4
--module-source-path	Передает структуру каталогов проекта	4.3.2
--module-version	Определяет версию модуля для компиляции	13.2.1
--patch-module	Расширяет компилируемый модуль с помощью классов во время компиляции	7.2.4
--processor-module-path	Указывает, где найти модули процессора аннотаций	4.2.1
--system	Переопределяет расположение системных модулей	
--upgrade-module-path	Объявляет месторасположение обновляемых модулей	6.1.3

Новый параметр --release

Вы когда-нибудь использовали параметры `-source` и `-target` при компиляции кода для запуска на более старой версии Java, только чтобы увидеть, как он завершается с ошибкой во время выполнения, поскольку вызов метода завершился неудачей с, казалось бы, необъяснимым сообщением? Может быть, это потому, что вы забыли указать `-bootclasspath`.

Без данной опции компилятор создает байт-код, который JVM с нужной версией понимает (хорошо), однако он связывается с API базовой библиотеки текущей версии (плохо). Это может приводить к вызовам типов или методов, которых не было в более старой версии JDK, и таким образом вызывать ошибки во время выполнения.

Начиная с Java 9, компилятор предотвращает эту распространенную ошибку с помощью параметра `--release`, который устанавливает правильное значение для всех трех других параметров.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

4.5. Упаковка модульного JAR-файла

На пути от идеи к запуску приложения следующий после написания кода и компиляции шаг — получение файлов классов и их упаковка в виде модуля. Как объясняется в разделе 3.1.2, это должно привести к *модульному JAR*, который похож на простой JAR, но содержит дескриптор модуля `module-info.class`. Следовательно, ожидается, что за упаковку будет отвечать доверенный инструмент `jar`. Ниже показано, как просто создать модульный JAR (в данном случае для *monitor.observer*):

```
$ jar --create
    --file mods/monitor.observer.jar
    -C monitor.observer/target/classes .
```

Если оставить в стороне новые наименования командной строки, то данный вызов работает точно так же, как и до Java 9. Интересная и неявная деталь — поскольку `monitor.observer/target/classes` содержит `module-info.class`, то же самое будет и в результате в `monitor.observer.jar`, что делает его модульным JAR.

Хотя инструмент `jar` работает так же, как и раньше, есть пара деталей и дополнений, связанных с модулем, например определение точки входа модуля, на которую нужно обратить внимание.

ПРИМЕЧАНИЕ

JAR не единственный формат, используемый для распространения байт-кода Java. JEE также работает с файлами WAR и EAR. Однако до тех пор, пока спецификация не будет обновлена, чтобы охватить модули, невозможно создавать модульные WAR или EAR-файлы.

4.5.1. Быстрый обзор jar

Дабы убедиться, что мы поняли друг друга, кратко рассмотрим, как `jar` используется для упаковки архивов. Несколько выше я отметил следующее: результатом является модульный JAR, если список включаемых файлов содержит дескриптор модуля `module-info.class`.

Возьмем команду, которая упаковывает *monitor.observer*, в качестве примера. В результате получается файл `module.observer.jar` в `mods`, который содержит все файлы классов из `monitor.observer/target/classes` и его подкаталогов. Поскольку классы содержат дескриптор модуля, JAR тоже будет содержать его и, следовательно, будет модульным JAR без каких-либо дополнительных усилий:

```
$ jar --create
    --file mods/monitor.observer.jar
    -C monitor.observer/target/classes
```

Этот режим отвечает за создание архива (альтернатива — -c)

Имя создаваемого файла архива (альтернатива — -f)

-C вносит изменения в jar в указанную папку, а точка (.) указывает, что в нее должны быть включены все исходные файлы

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Следует подумать о записи версии модуля с `--module-version` при его упаковке. В подразделе 13.2.1 объясняется, как это сделать.

4.5.2. Анализ JAR

При работе с JAR-файлами полезно знать способы анализа того, что вы создаете. Особенно важны файлы, которые лежат в JAR, и то, что находится в его дескрипторе модуля. К счастью, у `jar` есть варианты на оба случая.

Перечисление содержимого JAR

Самое очевидное, что нужно сделать, — посмотреть содержимое JAR, а это возможно с помощью `--list`. В следующем фрагменте кода показано содержимое файла `monitor.observer.jar`, созданного в предыдущем разделе. Он содержит папку `META-INF`, в которую мы не углубляемся, поскольку она существует уже много лет и не относится к системе модулей. Кроме того, есть дескриптор модуля и классы `DiagnosticDataPoint` и `ServiceObserver` в пакете `monitor.observer`. Ничего впечатляющего или неожиданного:

```
$ jar --list --file mods/monitor.observer.jar

> META-INF/
> META-INF/MANIFEST.MF
> module-info.class
> monitor/
> monitor/observer/
> monitor/observer/DiagnosticDataPoint.class
> monitor/observer/ServiceObserver.class
```

Это не новая команда — она выглядит иначе из-за новых наименований: `--list` — длинная версия `-t` и `-file` — длинная версия `-f`. До Java 9 `jar -t -f some.jar` делал бы то же самое.

Изучение дескриптора модуля

Дескриптор модуля является файлом класса и, таким образом, состоит из байт-кода. Это приводит к необходимости использовать инструменты для просмотра его содержимого. К счастью, `jar` может сделать это с помощью `--describe-module` (или `-d`). Изучая `monitor.observer.jar`, вы заметите, что это модуль с именем *monitor.observer*, который экспортирует пакет с тем же именем и требует наличия базового модуля:

```
$ jar --describe-module --file mods/monitor.observer.jar

> monitor.observer jar:.../monitor.observer.jar/!module-info.class
> exports monitor.observer
> requires java.base mandated
```

(Если вам интересно, что означает `mandated`, то вспомните, как в подразделе 3.1.4 было сказано: для каждого модуля неявно требуется базовый модуль, что говорит о наличии модуля *java.base*.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

4.5.3. Объявление точки входа

Чтобы запустить приложение Java, необходимо знать точку входа, которая является одним из классов, содержащих метод `public static void main(String[])`. Класс, содержащий этот метод, можно либо указать в командной строке при запуске приложения, либо записать в файл манифеста, который поставляется вместе с JAR. Не беспокойтесь, если точно не знаете, как работают один или оба этих параметра, поскольку в Java 9 добавлен третий вариант, подходящий для модулей.

Когда `jar` используется для упаковки файлов классов в архив, можно определить основной класс с помощью `--main-class ${класс}`, где `${класс}` — полное имя (то есть имя пакета с точкой и имя класса) класса с методом `main`. Он будет записан в дескрипторе модуля и по умолчанию станет использоваться в качестве основного класса в начальном модуле для запуска приложения (см. подробности в разделе 5.1).

ПРИМЕЧАНИЕ

Если вы привыкли включать запись Main-Class в манифест для создания исполняемых JAR-файлов, то вам будет приятно узнать, что jar --main-class делает то же самое.

Приложение *ServiceMonitor* имеет единственную точку входа в `monitor.Main`. Используйте `--main-class monitor.Main`, чтобы указать это во время упаковки:

```
$ jar --create
    --file mods/monitor.jar
    --main-class monitor.Main
    -C monitor/target/classes .
```

Применяя `--describe-module`, вы можете заметить, что основной класс был записан в дескрипторе:

```
$ jar --describe-module
    --file mods/monitor.jar

> monitor jar:.../monitor.jar/!module-info.class
# директивы requires и их содержимое опущены
> main-class monitor.Main
```

Интересно, что у инструмента `jar` нет ни возможностей, ни ответственности, чтобы проверить ваше утверждение о наличии такого класса. Там нет проверки того, существует ли он или содержит ли подходящий метод `main`. Если что-то пойдет не так, то ошибка не выведется, но запустить модуль не удастся.

4.5.4. Параметры архиватора

Мы изучили только самые важные варианты, которые может предложить `jar`. Парочка других опций становятся интересными в различных контекстах и объясняются в соответствующих главах. Чтобы их точно было легко найти, в табл. 4.2 перечислены опции, имеющие отношение к системе модулей. Посетите docs.oracle.com/javase/9/tools/jar.htm для получения официальной документации по `jar`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Таблица 4.2. Таблица всех параметров архиватора в алфавитном порядке, относящихся к модулю (команда jar). Описания основаны на документации, а ссылки указывают на разделы в данной книге, в которых подробно объясняется, как использовать параметры

Опция	Описание	Ссылка
--hash-modules	Записывает хеши зависимых модулей	
--describe-module, -d	Показывает имя модуля, зависимости, экспорт, пакеты и др.	4.5.2
--main-class	Точка входа в приложение	4.5.3
--module-path, -p	Указывает, где находятся модули приложения для записи хешей	3.4
--module-version	Определяет версию компилируемых модулей	13.2.1
--release	Создает JAR для нескольких выпусков, содержащий байт-код для различных версий Java	Приложение Д
--update	Обновляет существующий архив, например добавляя больше файлов классов	9.3.3

Резюме

- Убедитесь, что вы выбрали структуру каталогов, которая соответствует требованиям вашего проекта. Если сомневаетесь, то придерживайтесь структуры системы сборки по умолчанию.
- Команда `javac` для компиляции всех исходников модуля, в том числе объявления, такая же, как и в Java 9, за исключением того, что она использует путь модуля вместо пути класса.
- Исходный путь модуля (`--module-source-path`) информирует компилятор о том, как структурирован проект. Это поднимает операцию компилятора с обработки типов до обработки модулей, позволяя компилировать выбранный модуль и все его зависимости с помощью простой опции (`--module` или `-m`) вместо перечисления исходных файлов.
- Модульные JAR — простые JAR с дескриптором модуля `module-info.class`. Инструмент `jar` обрабатывает их так же, как и другие файлы классов, поэтому их упаковка в JAR не требует новых опций.
- При желании `jar` позволяет указать точку входа модуля (с `--mainclass`), которая является классом с методом `main`. Это упрощает запуск модуля.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

5 Запуск и отладка модульных приложений

В этой главе

- Запуск модульного приложения с указанием начального модуля.
- Загрузка ресурсов из модулей.
- Проверка модулей, наборов модулей и модульных графов.
- Сокращение и перечисление совокупности обозреваемых модулей.
- Отладка модульного приложения с ведением журнала.

Итак, у нас есть модули, определенные, скомпилированные и упакованные в модульные JAR, как описано в главах 3 и 4. Наконец пришло время включить JVM и запускать приложения с помощью команды `java`. Это позволяет нам обсудить концепцию времени выполнения: как загружать ресурсы из модулей (см. раздел 5.2). Однако рано или поздно что-то пойдет не так, поэтому в разделе 5.3 мы также рассмотрим отладку конфигурации модуля с помощью различных параметров командной строки.

К концу главы вы сможете запустить приложение, состоящее из модулей. Кроме того, будете иметь четкое представление о процессе обработки системой модулей заданной конфигурации и как это можно наблюдать с помощью ведения журнала и других диагностических инструментов.

На этом также завершается часть I, рассказывающая обо всем, что необходимо знать для написания, компиляции и запуска простых модульных приложений. Она закладывает основу для более сложных функций, которые будут рассмотрены в частях II и III, и главные среди этих функций — те, что поддерживают постепенный переход к системе модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

5.1. Запуск JVM с модулями

После всех сборок — определения зависимостей модуля и API, создания модульных JAR-файлов и размещения их в пути модуля — запуск JVM с модульным приложением станет до неприличия простым. Все, что нужно сделать, — это указать начальный модуль и, возможно, основной класс.

Команда `java` имеет опцию `--module ${`*`модуль`*`}`, которая указывает начальный модуль *${модуль}*. Отсюда начинается распределение модуля, и это также модуль, из которого будет запущен основной класс, то есть класс с методом `public static void main`.

Каждый конкретный класс либо определяется дескриптором начального модуля, либо указывается с помощью `--module ${`*`модуль`*`}/${`*`класс`*`}` путем добавления имени модуля с косой чертой и полного имени класса (см. подраздел 5.1.1).

Для приложения *ServiceMonitor* вся подготовка завершается вызовом, который вам уже знаком, запускающим JVM с *monitor* в качестве исходного модуля:

```
$ java
    --module-path mods:libs
    --module monitor
```

Как обсуждалось в разделе 3.4, `--module-path mods:libs` информирует систему модулей о том, что каталоги `mods` и `libs` содержат модули приложения *ServiceMonitor*. Опция `--module monitor` определяет *monitor* как начальный модуль, в результате чего система модулей распределит все зависимости *monitor* и построит модульный граф, как было описано выше. Затем она запустит основной класс, который был установлен в дескрипторе модуля во время упаковки в подразделе 4.5.3: `monitor.Main`.

5.1.1. Определение основного класса

Опция `--module` может использоваться также для определения основного класса приложения. Для этого после имени исходного модуля ставится косая черта и указывается полное имя класса (имя пакета, за которым следуют точка и имя класса).

Здесь вы явно определяете, что приложение запускается путем вызова метода `main` в классе `monitor.Main` модуля *monitor*:

```
$ java
    --module-path mods:libs
    --module monitor/monitor.Main
```

Указание основного класса в командной строке переопределяет все то, что было указано в дескрипторе модуля. Это значит, что приложения могут иметь несколько точек входа, как и без системы модулей. Если одна из них является подходящим значением по умолчанию, то имеет смысл включить ее в дескриптор модуля, как описано в подразделе 4.5.3.

Если *monitor* определяет `monitor.Main` в качестве основного класса, но по какой-то причине вы не хотите его использовать, то можно легко переопределить его.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

С помощью следующей команды приложение запускается путем вызова в *monitor* `some.other.MainClass`, игнорируя все, что определено в дескрипторе *monitor*:

```
$ java
    --module-path mods:libs
    --module monitor/some.other.MainClass
```

Чтобы все работало, начальный модуль должен содержать указанный класс. Поскольку это не относится к *monitor* и `some.other.MainClass`, выполнение только что показанной команды приводит к ошибке:

```
> Error: Could not find or load main class
>     some.other.MainClass in module monitor
```

5.1.2. Что, если начальный модуль и основной модуль — не один и тот же

Как поступить, если модуль, который вы хотите использовать в качестве начального, не содержит основной класс приложения? Сначала это кажется странной проблемой; но в разработке программного обеспечения только такие и встречаются — так что это не означает невозможности подобного события.

В качестве примера представим приложение для ПК, которое можно запустить в нескольких режимах (ввод данных, оценка, администрирование) и задать режим при запуске, выбрав правильный основной класс. Приложение сложное и состоит из множества модулей, и каждый режим имеет собственный модуль (*data.entry*, *data.evaluation*, *administration*), также содержащий соответствующую точку входа. Сверху находится *app*, который зависит от всех модулей приложения (на рис. 5.1 показан модульный граф).

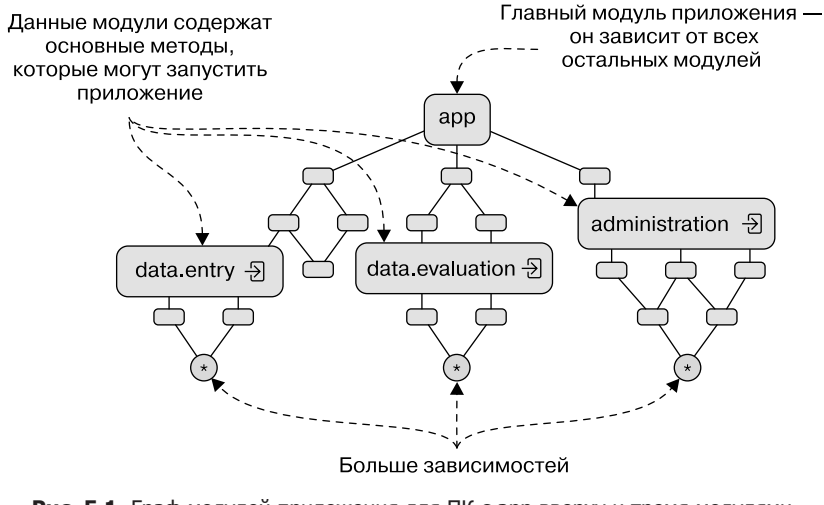


Рис. 5.1. Граф модулей приложения для ПК с app вверху и тремя модулями, содержащими точки входа, внизу

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Чтобы запустить данное приложение, следует использовать `--module app`, а затем указать основной класс из одного из других модулей — но возможно ли это? Решить проблему нам поможет терминология для двух задействованных модулей:

- есть модуль, зависящий от всех модулей (в переходном смысле), которые нужны приложению, — я назову его *all*;
- затем есть модуль, содержащий основной класс, который нужно запустить, — я назову его *main*.

До сих пор оба модуля всегда были одинаковыми, поэтому вы передали имя модуля в `--module`, сделав его начальным. Что делать, если это два отдельных модуля?

Суть в том, что система модулей непреклонна относительно происхождения основного класса. Не существует способа обмануть ее в поиске любого модуля, кроме начального. Следовательно, вы должны выбрать *main* в качестве исходного модуля, передав его в `--module`.

Предположительно этот подход не распределяет все зависимости правильно — и как тогда убедиться, что ни одна зависимость *all* не пропущена? На данном этапе становится полезной опция `--add-modules`, представленная в подразделе 3.4.3. С ее помощью можно определить *all* как дополнительный корневой модуль и заставить систему модулей также распределять его зависимости:

```
$ java
    --module-path mods
    --add-modules all
    --module main
```

Для приложения на ПК это означает следующее: вы всегда используете опцию приложения `--add-modules` с целью убедиться, что граф содержит все необходимые модули, а затем выбираете модуль для нужного режима в качестве начального. Например:

```
$ java
    --module-path mods
    --add-modules app
    --module data.entry
```

Кстати, если вам интересно, почему модули для различных режимов не зависят от всех необходимых модулей, то вот как минимум три ответа:

- приложение может быть отделено с помощью сервисов, как показано в главе 10, и *app* является потребителем;
- модули режима могут иметь некоторые необязательные зависимости, как описано в разделе 11.2, и *app* гарантирует присутствие их всех;
- помните, я говорил, что это странный случай?

5.1.3. Передача параметров приложению

Передать параметры в приложение так же просто, как и раньше. JVM помещает все после начального модуля в массив строк (разбитых по пробелам) и передает их в метод `main`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Предположим, вы вызываете *ServiceMonitor* следующим образом. Как вы думаете, что будет передано в `Main::main`? (Осторожно, это вопрос с подвохом!)

```
$ java
    --module-path mods:libs
    --module monitor
    --add-modules monitor.rest
    opt arg
```

Подвох вот в чем: `--add-modules monitor.rest` выглядит чем-то, за что должна отвечать система модулей. И так бы и было, если бы параметр находился в том же месте, что и раньше, — *до* `--module`. В существующем варианте он указывается после `--module`, заставляя JVM интерпретировать его как опцию для приложения и передавать дальше.

Для демонстрации расширим `Main::main` для вывода параметров:

```
public static void main(String[] args) {
    for (String arg : args) {
        System.out.print(arg + " / ");
    }

    // [...]

}
```

И действительно, получится вывод `--add-modules/monitor.rest/opt/arg`.

Будьте осторожны, делая `--module` последним параметром, который вы хотите обработать с помощью JVM, и оставляя все параметры приложения за ним.

5.2. Загрузка ресурсов из модулей

Раздел 3.3 подробно описывает, как правила доступности системы модулей обеспечивают надежную инкапсуляцию в пределах границ модуля. В разделе обсуждаются только типы, а во время выполнения вам, как правило, требуется и доступ к ресурсам. Будь то конфигурации, переводы, мультимедийные файлы или в некоторых случаях даже простые файлы `.class`, обычно их загружают из JAR-файлов, поставляемых с проектом. Поскольку JPMS превращает модульные JAR-файлы в модули, которые призваны строго инкапсулировать свои внутренние компоненты, необходимо изучить, как это повлияет на загрузку ресурсов. Прежде чем углубляться в тему, в следующих разделах я кратко расскажу о том, как ресурсы загружались в прошлом, и укажу на изменения, произошедшие в Java 9+. Затем мы подробнее рассмотрим загрузку ресурсов пакета в пределах границ модуля.

ПРИМЕЧАНИЕ

Тема доступа к ресурсам раскрывается в книге несколько раз: в разделе 6.3 объясняется, как получить доступ к ресурсам JDK, а в подразделе 8.2.1 рассматривается доступ к немодульным ресурсам. Для практической демонстрации того, как загружать ресурсы, обратитесь к ветке feature-resources приложения ServiceMonitor.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

5.2.1. Загрузка ресурсов до Java 9

Из-за отсутствия каких-либо границ между файлами JAR до 9-й версии Java каждому классу предоставляется доступ ко всем ресурсам в пути класса. Это даже хуже, чем доступность для типов, поскольку те по крайней мере могут использовать видимость пакета, чтобы перестать быть видимыми в нем. Для ресурсов это не работает.

ВАЖНАЯ ИНФОРМАЦИЯ

Чтобы загрузить ресурс, нужно вызвать getResource или getResourceAsStream из Class или ClassLoader. Концептуально эти методы почти идентичны: в них передается имя файла ресурса в виде строки, и они возвращают URL или InputStream, если находят его; в противном случае возвращается null. Чтобы не усложнять, мы будем использовать обозначение Class::getResource.

В листинге 5.1 показано, как загружать различные ресурсы. Пока все классы и ресурсы находятся в пути класса, не имеет значения, в каком JAR они находятся. На рис. 5.2 показан один JAR, который содержит все загруженные ресурсы, — если он находится в пути класса, то каждый вызов `Class::getResource` возвращает экземпляр `URL`.

Листинг 5.1. Загрузка ресурсов прошла успешно, поскольку все они находятся в пути класса

```
Class<?> anchor = Class
    .forName("monitor.resources.opened.Anchor")

URL pack = anchor.getResource("file.txt");

URL root = anchor.getResource("/file.txt");

URL meta = anchor.getResource("/META-INF/file.txt");

URL bytecode = anchor.getResource("Anchor.class");
```

Для вызова Class::getResource необходим экземпляр Class — два других класса Anchor также будут работать

Определяется как относительный путь к пакету, содержащему Anchor

Определяется как абсолютный путь из корня JAR из-за /, стоящего в начале

META-INF доступен через абсолютный путь

Загружает байт-код Anchor

5.2.2. Загрузка ресурсов, начиная с Java 9 и позже

Вы можете удивиться, почему в листинге 5.1 приведено так много разных примеров. Одни работают с модулями, а другие — нет, и я хочу обсудить каждый из них по очереди. Прежде чем мы подойдем к этому, рассмотрим различные API ресурсов в Java 9:

- методы в `Class` — хороший способ загружать ресурсы из модулей, мы кратко рассмотрим их поведение;
- методы в `ClassLoader` имеют другое и, как правило, менее полезное поведение, когда речь идет о модулях, и мы не будем их обсуждать. Если вы хотите использовать их, то обратитесь к их Javadoc;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

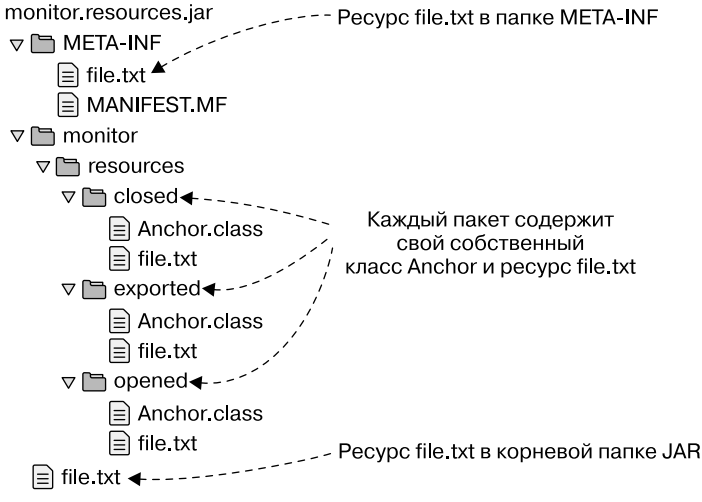


Рис. 5.2. Модуль monitor.persistence JAR содержит несколько ресурсов — по случайности те, которые требуются в листинге 5.1

- новый класс `java.lang.Module`, который мы подробно рассмотрим в разделе 12.3.3, также имеет метод `getResourceAsStream`. Он ведет себя так же, как и метод из `Class`.

После всего вышеперечисленного мы можем перейти к использованию рабочей лошадки `Class::getResource` для загрузки различных видов ресурсов в листинге 5.1 из модулей. Первое важное наблюдение заключается в том, что в одном и том же модуле каждый вызов возвращает экземпляр `URL`, то есть все ресурсы найдены. Это верно независимо от того, какие пакеты инкапсулирует модуль. Когда дело доходит до загрузки ресурсов за пределами границ модуля, все немного по-другому:

- ресурсы из пакета по умолчанию инкапсулированы (подробнее см. в подразделе 5.2.3);
- ресурсы из корня JAR или из папок, имена которых не могут быть сопоставлены с пакетами (например, `META-INF` из-за тире), никогда не инкапсулируются;
- файлы `.class` никогда не инкапсулируются;
- если ресурсы инкапсулированы, то вызов `getResource` возвращает `null`.

Причина, по которой большинство форм доступа не инкапсулированы, заключается в простоте миграции. Многие критически важные и широко используемые инструменты и фреймворки в экосистеме Java полагаются на конфигурации в корневом каталоге JAR или в папке `META-INF` (например, реализации JPA) или сканирование файлов `.class` (скажем, для поиска аннотированных классов). Если все ресурсы инкапсулированы по умолчанию, то эти инструменты могут по умолчанию не работать с модулями.

В то же время преимущества строгой инкапсуляции ресурсов гораздо менее значительны, чем то же самое для типов, вследствие чего было принято решение инкапсулировать ресурсы только в пакетах. Посмотрим, как это можно обойти.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

5.2.3. Загрузка ресурсов за пределами модуля

Всякий раз, когда методу `Class::getResource` или любому из подобных ему поручается загрузить ресурс, он проверяет, соответствует ли путь имени пакета. Проще говоря, если удалить имя файла из пути, а затем заменить все слеши на точки и получится правильное имя пакета — значит, ресурс загружается из пакета.

Выберем несколько строк из листинга 5.1 в качестве примеров. Вызов `anchor.getResource("file.txt")` сообщает JVM, что нужно загрузить ресурс `file.txt`, относящийся к классу `anchor`. Поскольку класс находится в пакете (в нашем примере это `monitor.resources.open`), ресурс загружается из данного пакета.

Обратный пример — `anchor.getResource("/META-INF/file.txt")`. Знак / в начале указывает на абсолютный путь (поэтому не имеет значения, в каком пакете находится `anchor`), и попытка преобразовать его в имя пакета приведет к получению строки `META-INF`. Это недопустимо в Java, и, следовательно, ресурс не будет загружен из пакета.

> **Открытие пакета**
>
> Важно понять, как JVM определяет, находится ли ресурс в пакете, ведь если это так, то он строго инкапсулирован. Кроме того, выражения `exports` *не дают* доступ к ресурсам. Поскольку `getResource` связан с API рефлексии, необходим другой механизм.
>
> Мы еще не обсуждали это, но то, что нужно для предоставления доступа к ресурсу, — выражение `opens`. Синтаксически оно работает точно так же, как `exports`, но дает только рефлексивный доступ к пакету, и это делает его идеальным для такого случая использования.

Об `opens` можно узнать гораздо больше, и в разделе 12.2 оно обсуждается подробно, но все, что нужно знать сейчас, — оно предоставляет доступ к ресурсам в инкапсулированных пакетах. Попробуем и создадим модуль *monitor.resources* около ресурсов, загруженных в листинге 5.1. Вот объявление модуля:

```
module monitor.resources {
    exports monitor.resources.exported;
    opens monitor.resources.opened;
}
```

Сравнив его с рис. 5.2, вы можете увидеть, что из трех пакетов один инкапсулирован, второй экспортирован, а третий открыт. Что произойдет, если запустить код из листинга 5.2?

Все зависит от модуля, который выполняет код. Если это *monitor.resources*, то вызовы проходят, поскольку инкапсуляция работает лишь в пределах границ модуля. Если код запускается каким-либо другим модулем, то ему доступен только для просмотра пакет `monitor.resources.opened`. Следовательно, `getResource` будет возвращать лишь отличный от `null` URL для `opened`, тогда как тот станет возвращать `null` для загрузки ресурсов из `closed` и `exported`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Другие вызовы из листинга 5.1 — `getResource("Anchor.class")`, `getResource("/file.txt")` и `getResource("/META-INF/file.txt")` — будут выполнены, поскольку они загружают любой байт-код или ресурсы, которых нет в пакетах. Как обсуждалось в подразделе 5.2.2, они не инкапсулированы.

Листинг 5.2. Загрузка ресурсов из пакетов с различной доступностью

```
URL closed = Class
    .forName("monitor.resources.closed.Anchor")
    .getResource("file.txt");
URL exported = Class
    .forName("monitor.resources.exported.Anchor")
    .getResource("file.txt");
URL opened = Class
    .forName("monitor.resources.opened.Anchor")
    .getResource("file.txt");
```

Ресурс не загрузится из инкапсулированного пакета

Ресурс не загрузится из экспортированного пакета

Ресурс загрузится из открытого пакета

Таким образом, чтобы предоставить доступ к ресурсам в пакете модуля, нужно сделать пакет открытым.

Открытие пакетов для предоставления доступа к ресурсам сделает другой код зависимым от внутренней структуры модуля. Чтобы избежать этого, рассмотрите возможность предоставить тип в публичном API, которому можно поручить загрузку ресурсов. После этого можно будет свободно перераспределять ресурсы по своему усмотрению, не вовлекая другие модули.

ПРИМЕЧАНИЕ

Если вы хотите избежать зависимостей от модуля, содержащего ресурсы, то создайте вместо него сервис. В главе 10 сервисы представлены подробно, и их использование для доступа к ресурсам было бы простым, если бы не приходилось разрешать конфликты имен. К счастью, для этого трудоемкого процесса есть отличная документация, так что я не буду повторять данный материал здесь. Проверьте Javadoc для ResourceBundleProvider, но убедитесь, что читаете версию хотя бы для Java 10 — она работает так же, как на Java 9, но документация более понятна: mng.bz/G28M.

5.3. Отладка модулей и модульных приложений

Система модулей решает сложную проблему и имеет амбициозные цели. Я думаю, она делает хорошую работу, упрощая использование не очень сложных случаев. Но не будем обманываться: это сложный механизм и что-то может пойти не так, особенно если вы перейдете к следующим двум частям данной книги, в которых рассматриваются миграция к системе модулей и ее более продвинутые возможности. В таких случаях может быть полезно взглянуть на внутреннюю работу системы модулей. К счастью, существует несколько способов сделать это:

- анализ и проверка модулей;
- тестирование построения модульного графа;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- изучение совокупности обозреваемых модулей;
- исключение модулей во время разрешения;
- запись поведения системы модулей.

В следующих разделах я представлю каждый из них по очереди.

5.3.1. Анализ отдельных модулей

Вы уже знаете, что `jmod describe` показывает модульные свойства JMOD (см. подраздел 3.1.1), а `jar --describe-module` выполняет аналогичную работу для JAR (см. подраздел 4.5.2). Это отличные способы изучения отдельных артефактов. Немного другой путь к тому же месту назначения предоставляет `java --describe-module`. После этого параметра, следующего за именем модуля, выводится путь к соответствующему артефакту, а также дескриптор модуля. Система модулей больше ничего не делает, не разрешает модули и не запускает приложение.

Так что, если `jmod description` и `jar --describe-module` работают с артефактами, то `java --describe` работает с модулями. В зависимости от ситуации тот или иной может быть более удобным, но в итоге их выводы схожи.

Еще раз обратившись к *ServiceMonitor*, вы можете использовать `--describe-module` для просмотра дескрипторов его модулей, а также платформенных модулей:

```
$ java
    --module-path mods
    --describe-module monitor.observer

> monitor.observer file:...monitor.observer.jar
> exports monitor.observer
> requires java.base mandated

$ java
    --module-path mods
    --describe-module java.sql

> java.sql@9.0.4
> exports java.sql
> exports javax.sql
> exports javax.transaction.xa
> requires java.base mandated
> requires java.logging transitive
> requires java.xml transitive
> uses java.sql.Driver
```

5.3.2. Проверка набора модулей

Просмотр отдельных модулей полезен для анализа известных проблем. Но как насчет неизвестных неполадок? Свободен ли путь модуля от дубликатов? Разделяются ли пакеты какими-либо модулями?

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Параметр `java --validate-modules` проверяет путь модуля на наличие ошибок. Он сообщает о дублированных модулях и разделенных пакетах, однако не строит граф, вследствие чего не может обнаружить отсутствующие модули или циклы зависимостей. На этом выполнение проверок `java` заканчивается.

Для примера я создал модуль *monitor.rest*, который содержит пакетный `monitor.observer`, как и модуль *monitor.observer*. Вот результат их проверки:

```
$ java
    --module-path mods
    --validate-modules

# опущены стандартизированные модули Java
# опущены нестандартизированные модули JDK
> file:.../monitor.rest.jar monitor.rest
> file:.../monitor.observer.beta.jar monitor.observer.beta
> file:.../spark.core.jar spark.core
> file:.../monitor.statistics.jar monitor.statistics
> file:.../monitor.jar monitor
> file:.../monitor.observer.jar monitor.observer
>     contains monitor.observer conflicts with module monitor.rest
> file:.../monitor.persistence.jar monitor.persistence
> file:.../monitor.observer.alpha.jar monitor.observer.alpha
> file:.../hibernate.jpa.jar hibernate.jpa
```

Вывод сначала перечисляет все модули JDK, в которых нет ошибок, а затем переходит к модулям приложения. Перечисляются отсканированные файлы JAR и обнаруженные в них модули, а также пакет, разделенный между *monitor.rest* и *monitor.observer*.

5.3.3. Проверка модульного графа

С параметром `--dry-run` JVM выполняет полное разрешение модуля, включая построение графа модулей и утверждение надежной конфигурации, но затем останавливается непосредственно перед выполнением основного метода. Описанное может показаться не особенно полезным, но я считаю, что это не так. Использование `--dry-run` в команде, которая содержит ошибки и, следовательно, предотвращает запуск приложения, ничего не меняет. Но когда вы наконец все сделаете правильно, команда завершится и вы вернетесь в командную строку. Это позволяет быстро экспериментировать с параметрами командной строки, пока не получится правильный их набор, без зацикленного запуска и прерывания приложения.

В качестве примера ошибочной команды попробуем запустить *ServiceMonitor*, не указывая путь модуля. Как и ожидалось, ничего не выйдет, поскольку без места для поиска прикладных модулей система модулей не может найти начальный модуль *monitor*.

```
$ java --module monitor
> Error occurred during initialization of boot layer
> java.lang.module.FindException:
>     Module monitor not found
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Добавление `--dry-run` ничего не меняет:

```
$ java --dry-run --module monitor

> Error occurred during initialization of boot layer
> java.lang.module.FindException:
>     Module monitor not found
```

То же для команды, которая должна работать:

```
$ java
    --module-path mods:libs
    --dry-run
    --module monitor
```

Это ни к чему не приводит. Команда верна, и система модулей удовлетворена, так что завершается после разрешения модуля без каких-либо сообщений.

Вспомните материал из подраздела 5.1.2, что `--dry-run` должен предшествовать `--module`, даже если эта последовательность выглядит неприятно. И примечание для экспертов: в случае применения пользовательского загрузчика классов, пользовательского менеджера безопасности или агентов они будут запущены даже с `--dry-run`.

5.3.4. Перечисление обозреваемых модулей и зависимостей

Вы использовали параметр `--list-modules` в подразделе 3.1.1, где он перечислил все модули платформы в текущей среде выполнения с `java --list-modules`. Теперь, когда вы имеете лучшее представление о том, как работает система модулей, я могу рассказать, что он делает даже нечто большее.

Перечисление совокупности обозреваемых модулей

Параметр `--list-modules` выводит список обозреваемых модулей. Система модулей больше ничего не делает, не разрешает модули и не запускает приложение.

Как показано в подразделе 3.1.4, совокупность обозреваемых модулей состоит из модулей платформы (используются во время выполнения) и модулей приложения (находятся в пути модуля). Во время разрешения из этого набора выбираются модули для построения графа. Приложение никогда не может содержать модули, которые не перечислены в списке `--list-modules`. (Но обратите внимание: вполне возможно и очень вероятно, что многие обозреваемые модули не попадут в граф, поскольку они не запрашиваются ни одним из корневых модулей — даже транзитивно.)

При вызове `java --list-modules` вы поставили перед JVM задачу получить список всех наблюдаемых модулей. Поскольку путь модуля не указан, будут выведены только платформенные модули среды выполнения.

Рассмотрим менее простой пример и перечислим модули в папках `mods` и `libs` приложения *ServiceMonitor*:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
$ java
    --module-path mods:libs
    --list-modules

> spark.core
# опущены зависимости Spark
# опущены стандартизированные модули Java
# опущены нестандартизированные модули Java
> monitor
> monitor.observer
> monitor.observer.alpha
> monitor.observer.beta
> monitor.persistence
> monitor.rest
> monitor.statistics
> hibernate.jpa
# опущены зависимости Hibernate
```

Если данный код выполнится при обычной установке JDK, то вывод нас ошеломит, поскольку в нем будут перечислены около 100 платформенных модулей. Вдобавок он всегда будет содержать все модули из пути модуля. В целом они полезны, чтобы увидеть, из каких модулей может быть построен граф, но также затрудняют просмотр леса за деревьями. Есть способ ограничить вывод разумным подмножеством, и мы рассмотрим его дальше.

Перечисление переходных зависимостей

Одно интересное подмножество этого длинного списка обозреваемых модулей — переходные зависимости исходного модуля. К счастью, можно сократить список с помощью только одной опции `--limit-modules`. Через минуту я объясню, как именно это работает. Пока просто поверьте мне, когда я говорю, что в сочетании с `--list-modules` можно использовать этот параметр для вывода списка всех переходных зависимостей любого конкретного модуля.

Вот парочка экспериментов с некоторыми платформенными модулями:

```
$ java --limit-modules java.xml --list-modules

> java.base
> java.xml
$ java --limit-modules java.sql --list-modules

> java.base
> java.logging
> java.sql
> java.xml
$ java --limit-modules java.desktop --list-modules

> java.base
> java.datatransfer
> java.desktop
> java.prefs
> java.xml
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Вы можете видеть, что *java.xml* зависит только от *java.base*, модуль SQL использует возможности записи и XML, и даже *java.desktop*, включающий в себя все AWT, Swing, некоторые медиа-API и API JavaBeans, имеет, к нашему удивлению, зависимости (хотя и по нелестной причине — это огромный модуль, содержащий *много* функций).

Такой же подход можно использовать и для изучения модулей приложения. Он становится особенно полезным, когда приложение выходит за рамки нескольких модулей, поскольку в этом случае становится все сложнее запомнить их все.

Еще раз взглянем на *ServiceMonitor* и рассмотрим зависимости некоторых из его модулей:

```
$ java
    --module-path mods:libs
    --limit-modules monitor.statistics
    --list-modules

> java.base
> monitor.observer
> monitor.statistics
$ java
    --module-path mods:libs
    --limit-modules monitor.rest
    --list-modules

> spark.core
# опущены зависимости Spark
> java.base
> monitor.observer
> monitor.rest
> monitor.statistics
```

Сочетание `--limit-modules` и `--list-modules` показывает, что *monitor.statistics* зависит только от *monitor.observer* (и от вездесущего базового модуля) и *monitor.rest* подтягивает все зависимости Spark.

Пришло время рассмотреть, как работает `--limit-modules`.

5.3.5. Исключение модулей во время разрешения

Вы только что использовали `--limit-modules` с целью сократить вывод `--list-modules`. Как это работает? Учитывая, что `--list-modules` выводит совокупность обозреваемых модулей, `--limit-modules` явно ее ограничивает. И, поскольку данный параметр можно использовать для просмотра всех переходных зависимостей модуля, все они должны быть оценены. Соединив оба наблюдения, получим довольно целостное определение параметра.

Параметр `--limit-modules ${`*модули*`}` принимает список разделенных запятыми имен модулей. Он сокращает совокупность обозреваемых модулей до указанных модулей и их переходных зависимостей. Если параметры `--add-modules` (см. подраздел 3.4.3) или `--module` (см. раздел 5.1) используются вместе с `--limit-modules`,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

то модули, указанные для этих двух параметров, становятся обозреваемыми, *но без учета их зависимостей*!

Рассмотрим, как шаг за шагом система модулей оценивает параметры.

1. Начиная с модулей, указанных для `--limit-modules`, JPMS определяет все их переходные зависимости. Это обусловлено требованиями к надежной конфигурации, описанными в подразделе 3.2.1.
2. Если использовались `--add-modules` или `--module`, то JPMS добавляет указанные модули (но не их зависимости).
3. JPMS применяет полученный набор как совокупность обозреваемых модулей для любых дальнейших шагов (например, перечисления модулей или запуска приложения).

Пара экспериментов с `--limit-modules` должны прояснить, как именно это работает. Начнем с перечисления всех переходных зависимостей в *monitor.rest*:

```
$ java
    --module-path mods:libs
    --limit-modules monitor.rest
    --list-modules

> java.base
# чтобы не загромождать вывод,
# я опускаю пути к файлам
> monitor.observer
> monitor.rest
> monitor.statistics
> spark.core
```

Вы можете вернуться к рис. 2.4 (см. выше), чтобы убедиться в правильности этих зависимостей. Как думаете, что произойдет при попытке запустить приложение? Для этого необходимо заменить `--list-modules` на `--module monitor`:

```
$ java
    --module-path mods:libs
    --limit-modules monitor.rest
    --module monitor

> Error occurred during initialization of boot layer
> java.lang.module.FindException:
>     Module monitor.persistence not found,
>     required by monitor
```

Данный результат демонстрирует два аспекта работы `--limit-modules`:

- начальный модуль, указанный с помощью параметра `--module`, становится обозреваемым (в противном случае выведется сообщение о том, что *monitor* не найден);
- ни одна из зависимостей исходного модуля не станет обозреваемой (в противном случае запустится приложение).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Предположительно то же самое относится и к `--add-modules`, тогда что ожидается в результате добавления `add-modules monitor.persistence`?

- Поскольку *monitor.persistence* теперь становится обозреваемым, ошибка должна исчезнуть.
- Так как его зависимость *hibernate.jpa* не обозреваема, выведется сообщение об ошибке.

Попробуем это воспроизвести:

```
$ java
    --module-path mods:libs
    --limit-modules monitor.rest
    --add-modules monitor.persistence
    --module monitor

> Error occurred during initialization of boot layer
> java.lang.module.FindException:
>     Module monitor.observer.alpha not found,
>     required by monitor
```

Этот конкретный случай показан на рис. 5.3.

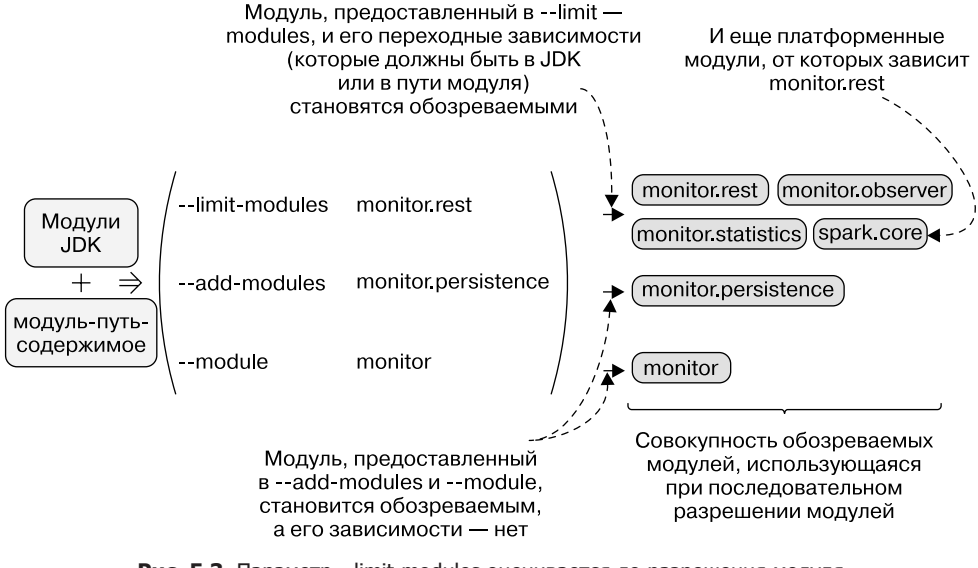


Рис. 5.3. Параметр --limit-modules оценивается до разрешения модуля

Черт возьми — реализации модуля *observer* также отсутствуют, поэтому мы никогда не узнаем о Hibernate. К счастью, нет ничего, что нельзя было бы решить с помощью дополнительных `--add-modules`:

```
$ java
    --module-path mods:libs
    --limit-modules monitor.rest
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
    --add-modules monitor.persistence,
        monitor.observer.alpha,monitor.observer.beta
    --module monitor

> Error occurred during initialization of boot layer
> java.lang.module.FindException:
>     Module hibernate.jpa not found,
>     required by monitor.persistence
```

То, что нужно!

В предыдущем подразделе использовалась вычисляемая совокупность для вывода списка всех модулей, ссылки на которые существуют, тем самым эффективно выводя все переходные зависимости какого-либо модуля. Это не единственный вариант применения `--limit-modules`. Более подробная информация появится, когда мы обсудим сервисы в главе 10.

5.3.6. Наблюдение за системой модулей с помощью ведения журнала

И последнее, но от этого не менее важное — мы подошли к чудодейственному средству отладки: к ведению журнала. Всякий раз, когда система плохо себя ведет и поиск проблем в очевидных местах (где бы ни возникало неправильное поведение) не приводит к каким-либо действиям, приходит время обратиться к журналу.

Если вы находитесь здесь, то, скорее всего, столкнулись с относительно редкой проблемой. В таком случае полезно знать способы извлечения сообщений журнала и связанную с ними информацию, а также то, как должен выглядеть журнал в лучшем случае — когда все работает. В данном разделе не показаны приемы устранения конкретной проблемы, но представлены инструменты для самостоятельного разрешения подобных задач.

Система модулей записывает сообщения, используя два разных механизма (потому что... а почему бы и нет?), один из которых более простой, а другой — более сложный для настройки:

- диагностические сообщения распознавателя;
- унифицированное ведение журнала JVM.

Мы рассмотрим оба варианта, начиная с более простого.

Диагностические сообщения во время разрешения модулей

С помощью параметра `--show-module-resolution` система модулей выводит сообщения во время разрешения модуля. Ниже приведен результат запуска приложения *ServiceMonitor* с данным параметром. Он идентифицирует корневые модули (единственный в данном случае), модули, которые были загружены в качестве зависимости, и то, какая это была зависимость:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
$ java
    --module-path mods:libs
    --show-module-resolution
    --limit-modules monitor
    --dry-run
    --module monitor
# для каждого модуля указывается его файл;
# я опустил его для краткости, но это может пригодиться
> root monitor
> monitor requires monitor.observer
> monitor requires monitor.rest
> monitor requires monitor.persistence
> monitor requires monitor.observer.alpha
> monitor requires monitor.observer.beta
> monitor requires monitor.statistics
> monitor.rest requires spark.core
> monitor.rest requires monitor.statistics
> monitor.persistence requires hibernate.jpa
> monitor.persistence requires monitor.statistics
> monitor.observer.alpha requires monitor.observer
> monitor.observer.beta requires monitor.observer
> monitor.statistics requires monitor.observer
# зависимости Spark опущены
# зависимости Hibernate опущены
```

Активизирует сообщения для разрешения модулей

Нам нужны только сообщения распознавателя, поэтому запуск приложения не требуется

По причинам, которые станут понятными при внедрении сервисов, необходимо ограничить совокупность обозреваемых модулей или распределить многие неожиданные модули

Экспериментировать с диагностическими сообщениями распознавателя из системы модулей довольно просто, но их нельзя настраивать. Пришло время перейти к более сложному и эффективному механизму.

Использование унифицированного ведения журнала для наблюдения за JPMS

В Java 9 появилась унифицированная архитектура ведения журналов, передающая множество сообщений, которое JVM генерирует с помощью одного и того же механизма. Приложение B представляет его и объясняет его настройку. Если вам не приходилось заниматься этим раньше, то загляните туда прямо сейчас. Я подожду здесь.

Отлично, вы вернулись. Вооружившись пониманием механизмов логирования и настраивания, можно более подробно рассмотреть, как работает система модулей. Все следующие эксперименты запускают приложение *ServiceMonitor* с известной командой, используя `--dry-run` для предотвращения фактического выполнения:

```
$ java
    --module-path mods:libs
    --dry-run
    --module monitor
```

Во фрагментах, приведенных ниже, показана лишь конфигурация `-Xlog`, используемая в дополнение к данной команде для определения выходных данных. Чтобы уменьшить шум и не позволить вам отвлекаться от главного, я удалил все

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

теги и вручную отредактировал сообщения с целью показать только самые важные части — настоящий журнал содержит гораздо больше информации.

Следуя собственному совету из приложения B, я заглянул в `-Xlog:help` и увидел тег `module`, который выглядит многообещающе. Я использовал его как `module*`, чтобы получить все сообщения, помеченные таким образом:

```
# -Xlog:module*

# опущено множество модулей
> java.base location: jrt:/java.base
> jdk.compiler location: jrt:/jdk.compiler
> spark.core location: file://...
> monitor.persistence location: file://...
> monitor.observer location: file://...
> monitor location: file://...
> monitor.rest location: file://...
> Phase2 initialization, 0.0977682 secs
```

Здесь система модулей рассказывает о загруженных модулях. Это все задействованные платформенные модули, а также модули *monitor.** и их зависимости. Чтобы получить более подробную информацию, включим отладочные сообщения:

```
# -Xlog:module*=debug

# Тьфу! Почти 1500 строк сообщений из журнала.
```

Данный вывод кажется ошеломляющим, но если пройти его шаг за шагом, то он становится несложным. Кроме того, у вас есть возможность увидеть в действии некоторые детали работы системы модулей. Итак, сделаем это!

Первое, с чем приходится иметь дело системе модулей, — это, что интересно, безымянный модуль. Он пока остается неразрешенной загадкой (см. раздел 8.2). Далее следует базовый модуль — как описано в подразделе 3.1.4, все остальные модули зависят от него, поэтому имеет смысл определить его на ранней стадии:

```
> recording unnamed module for boot loader
> java.base location: jrt:/java.base
> Definition of module: java.base
```

Затем начинается создание всех обозреваемых модулей:

```
> jdk.compiler location: jrt:/jdk.compiler
> creation of module: jdk.compiler
> jdk.localedata location: jrt:/jdk.localedata
> creation of module: jdk.localedata
> monitor.observer.alpha location: file://...
> creation of module: monitor.observer.alpha
# создание множества других модулей
```

После того как все модули созданы, система модулей обрабатывает их дескрипторы, добавляя границы чтения и экспорт пакетов, как было указано в них:

```
> Adding read from module java.xml to module java.base
> package com/sun/org/apache/xpath/internal/functions in module java.xml
>     is exported to module java.xml.crypto
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
> package javax/xml/datatype in module java.xml
>     is exported to all unnamed modules
> package org/w3c/dom in module java.xml
>     is exported to all unnamed modules
> Adding read from module monitor.statistics to module monitor.observer
> Adding read from module monitor.statistics to module java.base
> package monitor/statistics in module monitor.statistics
>     is exported to all unnamed modules
```

Вы можете заметить, что экспорт пакета выражается как `"to module ..."` и иногда значение даже отличается от `all unnamed modules`. В чем дело? Раздел 11.3 посвящен этому вопросу — сейчас достаточно признать, что экспорт пакетов все-таки обрабатывается.

Вот и все! Последнее сообщение, которое мы видели раньше, пришло незадолго до прерывания пробного запуска:

```
> Phase2 initialization, 0.1048592 secs
```

Если вы сделаете еще один шаг внутрь «Матрицы» и переключите уровень логирования на `trace`, то столкнетесь с несколькими тысячами сообщений, но никаких впечатляющих откровений там не окажется. Вы лишь заметите, что при загрузке каждого класса система модулей записывает, к какому пакету и модулю он принадлежит, прежде чем в конечном итоге определить пакеты. Как только это будет сделано, соответствующий модуль создастся.

Если удалить `--dry-run` и запустить приложение, то больше информации вы не получите. При отладке новые сообщения не создаются; и в `trace` просто отобразится процесс, как куча вложенных классов присваивается существующим пакетам.

ПРИМЕЧАНИЕ

Если вам интересно, то все описанное происходит в одном потоке. Это можно проверить, выведя идентификатор потока с помощью -Xlog:module*=debug:stdout:tid, который покажет один и тот же идентификатор для всех операций, связанных с модулем.

Теперь вы знаете, как настроить ведение журнала и как должен выглядеть журнал. Данные знания могут стать отличным инструментом диагностики. Это удобно, когда модульное приложение работает не по плану, а другие подходы не приводят к анализу, который помог бы решить проблему.

5.4. Параметры виртуальной машины Java

Как и компилятор и архиватор, виртуальная машина получила ряд новых параметров командной строки, которые взаимодействуют с системой модулей. Для вашего удобства они перечислены в табл. 5.1. Официальная документация находится по адресу docs.oracle.com/javase/9/tools/java.htm.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Таблица 5.1. Таблица всех параметров виртуальной машины, связанных с модулями, в алфавитном порядке (команда java). Описания основаны на документации, а ссылки указывают на разделы в этой книге, в которых подробно объясняется, как использовать данные параметры

Параметр	Описание	Ссылка
--add-exports	Позволяет модулю экспортировать дополнительные пакеты	11.3.4
--add-modules	Объявляет корневые модули в дополнение к начальному модулю	3.4.3
-add-opens	Позволяет модулю открывать дополнительные пакеты	12.2.2
--add-reads	Добавляет ребра чтения между модулями	3.4.4
--describe-module, -d	Показывает имя модуля, его зависимости, экспорты, пакеты и др.	5.3.1
--dry-run	Запускает виртуальную машину, но прерывает выполнение перед вызовом метода main	5.3.3
--illegal-access	Настраивает способ обработки доступа из пути класса к внутренним API JDK	7.1.4
--limit-modules	Ограничивает совокупность обозреваемых модулей	5.3.5
--list-modules	Перечисляет все обозреваемые модули	5.3.4
--module, -m	Определяет начальный модуль	5.1
--module-path, -p	Указывает, где находятся модули приложения	3.4
--patch-module	Расширяет компилируемый модуль с помощью классов во время компиляции	7.2.4
--show-module-resolution	Выводит сообщения во время разрешения модулей	5.3.6
--upgrade-module-path	Объявляет месторасположение обновляемых модулей	6.1.3
--validate-modules	Проверяет путь модуля на наличие ошибок	5.3.2

Помимо возможности использования этих параметров в командной строке, вы также можете указать некоторые из них в исполняемом манифесте JAR, задать их в определенной переменной среды, которую берет команда `java`, или поместить в файл аргументов, передаваемый при запуске JVM. Подраздел 9.1.4 объясняет все это.

Вы достигли второй вехи и завершения части I. Теперь вы хорошо разбираетесь в основах системы модулей. По возможности потратьте некоторое время на работу с тем, что узнали, — например, создайте собственную демоверсию или поэкспериментируйте с *ServiceMonitor* (github.com/CodeFX-org/demo-jpms-monitor). Что читать дальше, зависит от того, есть ли уже у вас проект, который нужно перенести на Java 9+, и, возможно, даже модульный (см. часть II), или же вы больше заинтересованы в изучении того, чем еще может быть полезна система модулей (см. часть III).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Резюме

- Начальный модуль определяется с помощью `--module`. Если он содержит основной класс, то больше не нужно запускать приложение; в противном случае полное имя класса добавляется к имени модуля после косой черты.
- Обязательно перечислите все параметры JVM до `-module`, иначе они будут рассматриваться как параметры приложения и не повлияют на систему модулей.
- Обозреваемые модули можно перечислить с помощью `--list-modules`. Это удобно, если нужно отладить проблемы и посмотреть, какие модули были доступны для распределения.
- Если используется `--limit-modules`, то совокупность обозреваемых модулей будет состоять только из указанных модулей и их переходных зависимостей, что уменьшает количество модулей, доступных во время разрешения. Вместе с `--list-modules` это отличный способ определить переходные зависимости модуля.
- Параметр `--add-modules` можно использовать для определения дополнительных корневых модулей, помимо исходного. Если модуль не является обязательным (например, потому, что доступен только с помощью рефлексии), необходимо использовать `--add-modules`, чтобы убедиться в его добавлении в граф.
- Параметр `--dry-run` запускает JVM и позволяет системе модулей обрабатывать настройки (путь модуля, исходный модуль и т. д.) и строить граф, но все завершается непосредственно перед вызовом метода `main`. Это позволяет проверить конфигурацию без запуска приложения.
- Система модулей регистрирует различные сообщения, которые могут быть выведены либо с помощью простого `--show-module-resolution`, либо с более сложным `-Xlog:module*`. Они позволяют проанализировать, как система составляет граф, что пригодится при устранении неполадок.
- Загрузка ресурсов из модулей работает так же, как загрузка их из JAR-файлов. Единственные исключения — ресурсы, которые не являются файлами `.class` и находятся в пакете другого модуля (в отличие, например, от корня JAR или папки `META-INF`). По умолчанию они инкапсулированы и потому недоступны.
- Модуль может использовать директивы `opens` для предоставления рефлексивного доступа к пакетам, который раскрывает расположенные в нем ресурсы и позволяет другим модулям загружать их. К сожалению, это решение предполагает, что другой код зависит от внутренней структуры модуля.
- При загрузке ресурсов по умолчанию используются методы `getResource` и `getResourceAsStream` из `Class` или их аналоги в новом типе `java.lang.Module`. Те же методы в `ClassLoader` обычно не настолько полезны.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Часть II

Адаптация под реальные проекты

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В части I этой книги были рассмотрены основы системы модулей и способы компиляции, сборки и запуска модульных приложений. Вдобавок к обучению соответствующим механизмам она показала, как будут организованы и разработаны грядущие Java-проекты.

Но как насчет существующих проектов? Я уверен, вы хотели бы, чтобы они работали на Java 9 или более поздней версии, возможно, даже в модульном виде. В данной части книги рассказывается, как это сделать.

Первый шаг, получение проекта для компиляции и запуска на Java 9+, обязателен для любой кодовой базы, если вы не хотите оставлять ее на Java 8 после окончания срока службы или платить за поддержку. Второй шаг, превращение артефактов проекта в модульные JAR-файлы, является необязательным и может быть выполнен со временем.

Главы 6 и 7 описывают переход на Java 9. Они посвящены тому, чтобы ваш немодульный проект на основе путей классов работал с новейшей версией (без создания каких-либо модулей). Затем, в главе 8, рассматриваются функции, которые позволяют постепенно наращивать модульность вашего проекта. Глава 9 дает некоторые стратегические советы о том, как перенести и модульно реализовать проект, используя то, что вы узнали в главах 6–8.

Я рекомендую читать главы в указанном порядке, но если вы предпочитаете изучать технические детали, только когда они вам нужны, то можете начать с главы 9. В качестве альтернативы можно прочитать о проблемах, с которыми вы, скорее всего, столкнетесь в первую очередь: речь идет о зависимостях от модулей JEE (см. раздел 6.1) и внутренних компонентах JDK (см. раздел 7.1).

Конкретных примеров для всего, что показано в этой части книги, нет. Репозиторий по адресу github.com/CodeFX-org/demo-java-9-migration содержит вариант приложения *ServiceMonitor* с большим количеством проблем, которые необходимо исправить, чтобы работать на Java 9+. Попробуйте!

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

6 Проблемы совместимости при переходе на Java 9 и выше

В этой главе

- Почему модули JEE устарели и не разрешены по умолчанию.
- Компиляция и запуск кода, который зависит от модулей JEE.
- Почему приведение к URLClassLoader не работает.
- Понимание нового макета образа среды выполнения JDK.
- Замена удаленного механизма расширения и механизма переопределения утвержденных стандартов и параметра пути к загрузочному классу.

Здесь и в главе 7 обсуждаются проблемы совместимости при миграции имеющейся кодовой базы на Java 9 и более поздние версии. Вы пока еще не будете создавать какие-либо модули; данные главы посвящены созданию и запуску существующего проекта в новейшей версии.

Почему переход на Java 9+ требует целых двух глав? Неужели нельзя установить новейшую версию JDK и ожидать, что все будет работать? Разве Java не предназначен для обратной совместимости? Да — если ваш проект, включая его зависимости, характеризуется только не устаревшим, стандартизированным, документированным поведением. Но это большое «если», и оказывается, в отсутствие какого-либо принуждения сообщество Java отклонилось от данного пути.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Как вы увидите в данной главе, система модулей сделала некоторые функции Java устаревшими, удалила другие и изменила кое-какие внутренние компоненты:

- модули, содержащие API JEE, устарели и требуют разрешения вручную (см. раздел 6.1);
- загрузчик классов приложения (также называемый загрузчиком классов системы) — это теперь не `URLClassLoader`, что нарушает некоторые приведения (см. раздел 6.2);
- макет каталога образа среды выполнения Java (JRE и JDK) был пересмотрен (см. раздел 6.3);
- ряд механизмов, таких как компактные профили и механизмы переопределения утвержденных стандартов, были удалены (см. раздел 6.4);
- также были изменены некоторые другие, незначительные детали, например, больше не допускается использование одинарного подчеркивания в качестве идентификатора (см. раздел 6.5).

И это еще не все. В главе 7 обсуждаются две дополнительные проблемы (внутренние API и разделенные пакеты). Для них предусмотрена собственная глава, поскольку, скорее всего, вы встретите их снова с модулями, отличными от JDK, уже после переноса своего проекта.

В совокупности эти изменения нарушают некоторые библиотеки, фреймворки, инструменты, методы и, возможно, ваш код, поэтому, к сожалению, обновление до Java 9+ не всегда проходит легко и просто. Вообще говоря, чем больше и старше проект, тем выше шанс, что придется поработать над ним. Опять же обычно это с пользой затраченное время, поскольку вы можете также погасить некоторые технические долги и привести кодовую базу в лучшую форму.

К концу этой и последующих глав вы узнаете о проблемах обновления до Java 9, 10 и 11 или даже более поздних версий. Взяв приложение, вы сможете сделать обоснованные предположения о том, что необходимо сделать; и при условии, что все зависимости совпадают, вы сможете заставить его работать в новейшей версии. Вы также будете хорошо подготовлены к главе 9, в которой рассматриваются стратегии перехода на Java 9 и более поздние версии.

О пути класса

Глава 8 может многое рассказать о том, как немодульный код работает на модульном JDK. На данный момент вам нужно знать только следующее:

- путь класса все еще полностью функционален. Во время перехода на Java 9+ вы продолжите использовать его вместо пути модуля;
- даже тогда система модулей все еще работает: например, в случаях разрешения модулей;
- код в пути к классам автоматически считывает большинство модулей (но не все, см. раздел 6.1), поэтому они доступны во время компиляции или выполнения без дополнительной настройки.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

6.1. Работа с модулями JEE

Большая часть кода в Java SE связана с Java EE / Jakarta EE (которые я сокращаю как JEE): на ум приходит CORBA, а также архитектура Java для привязки XML (JAXB) и API Java для веб-сервисов XML (JAX-WS). Эти и другие API оказались в шести модулях, показанных в табл. 6.1. Это могла быть не более чем маленькая дополнительная заметка, и так история бы и закончилась, но, к сожалению, все иначе. Когда вы пытаетесь скомпилировать или запустить код, который зависит от класса, находящегося в одном из этих модулей, система модулей будет утверждать, что модули отсутствуют в графе.

Таблица 6.1. Шесть модулей JEE. Описания взяты из документации

Имя модуля	Описание	Пакеты
java.activation	Определяет API JavaBeans Activation Framework (JAF)	javax.activation
java.corba	Определяет привязку Java к API CORBA Open Management Group (OMG) и API RMI-IIOP	javax.activity, javax.rmi, javax.rmi.CORBA, org.omg*
java.transaction	Определяет подмножество Java Transaction API (JTA) для поддержки взаимодействия с CORBA	javax.transaction
java.xml.bind	Определяет JAXB API	javax.xml.bind*
java.xml.ws	Определяет JAX-WS и Web Services Metadata API	javax.jws, javax.jws.soap, javax.xml.soap, javax.xml.ws*
java.xml.ws.annotation	Определяет подмножество Common Annotations API для поддержки программ, работающих на платформе Java SE	javax.annotation

Ниже представлена ошибка компиляции на Java 9 для класса, использующего `JAXBException` из модуля *java.xml.bind*:

```
> error: package javax.xml.bind is not visible
> import javax.xml.bind.JAXBException;
>                ^
>     (package javax.xml.bind is declared in module java.xml.bind,
>      which is not in the module graph)
> 1 error
```

Если вы справитесь с компилятором, но забудете поменять среду выполнения, то получится `NoClassDefFoundError`:

```
> Exception in thread "main" java.lang.NoClassDefFoundError:
>         javax/xml/bind/JAXBException
>     at monitor.Main.main(Main.java:27)
> Caused by: ClassNotFoundException:
>         javax.xml.bind.JAXBException
>     at java.base/BuiltinClassLoader.loadClass
>         (BuiltinClassLoader.java:582)
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
>     at java.base/ClassLoaders$AppClassLoader.loadClass
>         (ClassLoaders.java:185)
>     at java.base/ClassLoader.loadClass
>         (ClassLoader.java:496)
> ... 1 more
```

В чем дело? Почему правильно стандартизированные API Java не представлены в коде в пути класса и что с этим можно сделать?

6.1.1. Что особенного в модулях JEE?

Java SE содержит несколько пакетов, состоящих из *утвержденных стандартов* и *автономных технологий*. Эти технологии разрабатываются вне процесса сообщества Java (JCP), часто потому, что опираются на стандарты, регулируемые другими органами. Примерами являются объектная модель документа (DOM), созданная Консорциумом World Wide Web (W3C) и рабочей группой по разработке приложений для гипертекстовых веб-приложений (WHATWG), и Simple API для XML (SAX). Если вам это интересно, то полный список и соответствующие пакеты представлены на mng.bz/8Ek7. Многие из них непропорционально попадают в модули JEE, перечисленные в табл. 6.1: *java.corba*, *java.xml.bind* и *java.xml.ws*.

Исторически Java Runtime Environment (JRE) поставлялась с реализациями этих технологий, но была готова позволить пользователям обновлять их независимо от JRE. Это можно было сделать с помощью *механизма переопределения утвержденных стандартов* (см. подраздел 6.5.3).

Аналогично серверы приложений часто расширяют или обновляют API CORBA, JAXB или JAX-WS, а также JavaBeans Activation Framework (в *java.activation*) или JTA (в *java.transaction*), предоставляя собственные реализации. Наконец, *java.xml.ws.annotation* содержит пакет `javax.annotation`. Он часто расширяется различными реализациями JSR 305, которые наиболее известны своими аннотациями, связанными с `null`.

Во всех этих случаях расширения или замены API, поставляемых с Java, хитрость заключается в использовании точно таких же имен пакетов и классов, чтобы классы загружались из внешнего JAR вместо встроенных. На языке системы модулей это называется *разделенным пакетом*: один и тот же пакет разделяется на разные модули или модуль и путь класса.

Конец разделенных пакетов

Разделение пакетов больше не работает в Java 9+ и более поздних версиях. Мы рассмотрим это подробно в разделе 7.2, сейчас достаточно знать, что классы в пути к классам в пакете, распространяемом с Java, фактически невидимы:

- если Java содержит класс с таким же полностью определенным именем, то будет загружен;
- если встроенная версия Java пакета не содержит требуемый класс, то в результате появится ошибка компиляции или `NoClassDefFoundError`, которые я показал ранее. И это происходит независимо от того, присутствует ли класс в пути к классам.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Это общий механизм для всех пакетов во всех модулях: разделение их между модулем и путем класса делает часть пути класса невидимой. Особенными шесть модулей JEE делает то, что, в отличие от других модулей, их обычно расширяют или модернизируют с помощью подхода с разделением пакетов.

Чтобы серверы приложений и библиотеки, такие как реализации JSR 305, работали без расширенной конфигурации, был определен компромисс: для кода в пути классов Java 9 и 10 по умолчанию модули JEE не распределяются, то есть не помещаются в граф и, следовательно, недоступны (см. подраздел 3.4.3 для нераспределенных модулей и подраздел 8.2.2 для подробной информации о сценарии для пути к классам).

Это хорошо работает для приложений, поставляющихся с собственными реализациями API JEE, но не подходит для тех, которые опирались на варианты JDK. Не имея дальнейшей настройки, код в пути классов, использующий типы из этих шести модулей, не сможет скомпилироваться и выполниться.

Чтобы избавиться от этой сложности и должным образом отделить Java SE от JEE, эти модули объявили устаревшими в Java 9 и удалили в Java 11. После удаления такие инструменты командной строки, как `wsgen` и `xjc`, также больше не поставляются с JDK.

6.1.2. Разрешение модулей JEE вручную

Как поступить при ошибке компиляции или выполнения из-за отсутствия API JEE либо если анализ JDeps (см. приложение Г) показывает, что вы зависите от модулей JEE? Существуют три варианта решения.

- Если приложение работает на сервере, то способно обеспечить реализацию этих API, и в данном случае ошибки во время выполнения не появятся. В зависимости от настройки, возможно, придется исправлять ошибки сборки, хотя любое из двух других решений должно избавить вас от этой необходимости.
- Выберите стороннюю реализацию этого API и добавьте ее в качестве зависимости в свой проект. Поскольку модули JEE по умолчанию не разрешены, данная реализация будет использоваться без проблем во время компиляции и во время выполнения.
- В Java 9 и 10 добавьте платформенный модуль с `--add-modules`, как описано в подразделе 3.4.3. Поскольку модули JEE удалены в Java 11, там это не сработает.

Пример в начале раздела попытался использовать исключение `JAXBException` из модуля *java.xml.bind*. Вот как нужно сделать этот модуль доступным для компиляции с помощью `--add-modules`:

```
$ javac
    --class-path ${jar-файлы}
    --add-modules java.xml.bind
    -d ${выходной_каталог}
    ${исходные_файлы}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Когда код скомпилирован и упакован, необходимо снова добавить модуль для выполнения:

```
$ java
    --class-path ${jar-файлы}
    --add-modules java.xml.bind
    ${основной_класс}
```

Если вы зависите от нескольких API JEE, то может быть проще добавить весь модуль *java.se.ee* вместо каждого отдельного модуля. Таким образом, становятся доступными все шесть модулей EE, что немного упрощает работу. (Как это работает? Читайте о модулях-агрегаторах в подразделе 11.1.5.)

ВАЖНАЯ ИНФОРМАЦИЯ

Вместо --add-modules я настоятельно рекомендую рассмотреть возможность добавления сторонних реализаций необходимых API в качестве обычных зависимостей проекта. В подразделе 9.1.4 обсуждаются недостатки использования параметров командной строки, поэтому обязательно прочитайте его, прежде чем идти по данному пути. А поскольку модули JEE удалены в Java 11, рано или поздно вам все равно понадобится сторонняя реализация.

Усилие при ручном добавлении модулей JEE требуется только для немодульного кода. Став модульными, модули EE перестают быть особенными: вы можете запрашивать их как любой другой модуль, и они станут распределяться как любой другой модуль — по крайней мере до тех пор, пока не будут удалены.

Сторонние реализации JEE

Сравнение и обсуждение сторонних реализаций различных API JEE приведет к удалению от системы модулей, так что здесь я не стану этого делать. Для получения списка альтернатив см. JEP 320 (http://openjdk.java.net/jeps/320) или Stack Overflow (mng.bz/0p29).

6.1.3. Вмешиваемся в сторонние реализации модулей JEE

Возможно, вы использовали механизм переопределения утвержденных стандартов для обновления стандартов и автономных технологий. В этом случае можно задаться вопросом, что с ним случилось после прихода модулей. Как вы уже догадались, он был удален и заменен чем-то новым.

И компилятор, и среда выполнения предлагают опцию `--upgrade-module-path`, которая принимает список каталогов, отформатированный так же, как и для пути модуля. Создавая модульный граф, система модулей ищет в этих каталогах артефакты и использует их для замены обновляемых модулей. Шесть модулей JEE всегда являются обновляемыми:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- *java.activation*;
- *java.corba*;
- *java.transaction*;
- *java.xml.bind*;
- *java.xml.ws*;
- *java.xml.ws.annotation*.

Поставщики JDK могут сделать больше модулей обновляемыми. Например, в Oracle JDK это относится к *java.jnlp*. Кроме того, модули приложения, которые были связаны в образ с помощью `jlink`, всегда можно обновить (см. подраздел 14.2.1 для получения дополнительной информации).

JAR-файлы в обновленном пути модуля не обязательно должны быть модульными. Если у них нет дескриптора модуля, то они превратятся в автоматические модули (см. раздел 8.3) и все равно смогут заменить модули Java.

6.2. Приведение к URLClassLoader

При запуске проекта на Java 9 или более поздней версии можно столкнуться с ошибкой приведения класса, как показано в следующем примере. Здесь JVM жалуется, что не может привести экземпляр `jdk.internal.loader.ClassLoaders.AppClassLoader` к `URLClassLoader`:

```
> Exception in thread "main" java.lang.ClassCastException:
>     java.base/jdk.internal.loader.ClassLoaders$AppClassLoader
>     cannot be cast to java.base/java.net.URLClassLoader
>         at monitor.Main.getClassPathContent(Main.java:46)
>         at monitor.Main.main(Main.java:28)
```

AppClassLoader не расширяет URLClassLoader, поэтому во время приведения происходит ошибка

Класс, возвращаемый getClass, — это AppClassLoader

Что это за новый тип и почему он ломает код? Выясним! В процессе вы узнаете, как Java 9 изменяет поведение загрузки классов для повышения производительности запуска. Таким образом, даже если ваш проект не страдает от данной конкретной проблемы, это прекрасная возможность углубить знания Java.

6.2.1. Загрузчики классов приложения: тогда и сейчас

Во всех версиях Java *загрузчик классов приложения* (часто называемый *загрузчиком классов системы*) — один из трех загрузчиков классов, используемых JVM для запуска приложения. Он загружает классы JDK, которым не нужны никакие специальные привилегии, а также все классы приложения (если только приложение не задействует собственные загрузчики классов, в этом случае ничего из нижеперечисленного не применяется).

Вы можете получить доступ к загрузчику классов приложения, вызвав `ClassLoader.getSystemClassLoader()` или `getClass().GetClassLoader()` для экземпляра

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

одного из ваших классов. Оба метода обещают предоставить экземпляр типа `ClassLoader`. В Java 8 и более ранних версиях загрузчик классов приложения представляет собой `URLClassLoader`, подтип `ClassLoader`; и, поскольку `URLClassLoader` предлагает ряд методов, которые могут оказаться полезными, обычно экземпляр приводится к нему. Пример этого показан в листинге 6.1.

Листинг 6.1. Приведение загрузчика классов приложения к URLClassLoader

```
private String getClassPathContent() {
    URLClassLoader loader =
        (URLClassLoader) this.getClass().getClassLoader();
    return Arrays.stream(loader.getURLs())
            .map(URL::toString)
            .collect(joining(", "));
}
```

Получает загрузчик классов приложения и приводит его к URLClassLoader

getURLs не существует в ClassLoader, что и является причиной приведения

Без модулей в качестве представления JAR во время выполнения `URLClassLoader` понятия не имеет, в каком артефакте найти класс; вследствие чего всякий раз при необходимости загрузить класс `URLClassLoader` сканирует каждый артефакт в пути к классу, пока не найдет искомое (рис. 6.1). Очевидно, что это довольно неэффективный способ.

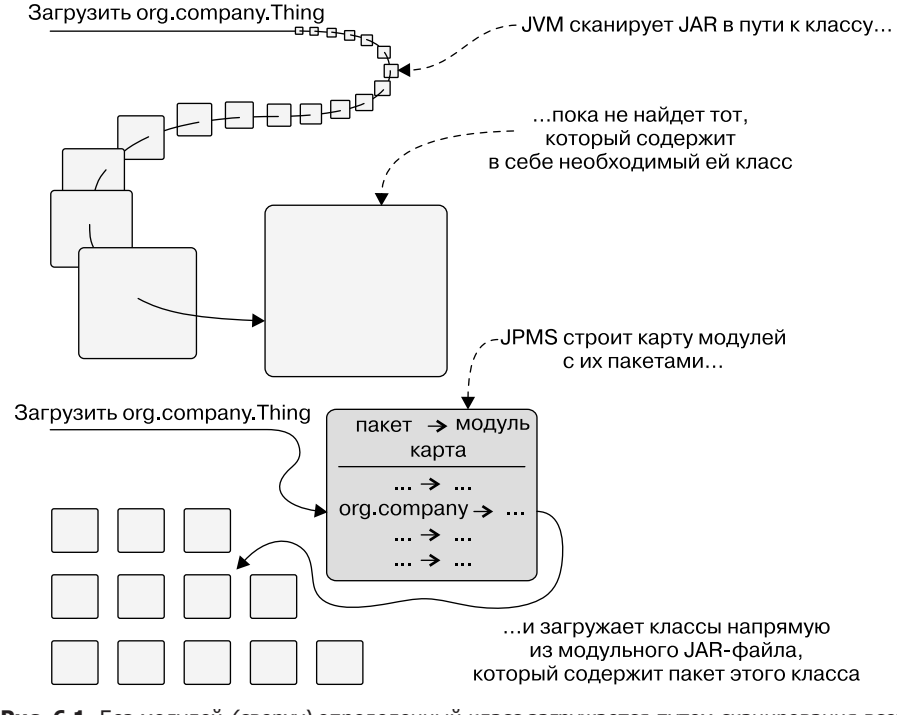


Рис. 6.1. Без модулей *(сверху)* определенный класс загружается путем сканирования всех артефактов в пути к классам. С модулями *(снизу)* загрузчик классов знает, из какого модульного JAR-пакета приходит пакет, и загружает его прямо оттуда

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Теперь перейдем к Java 9+. С JAR-файлами, получающими надлежащее представление во время выполнения, поведение загрузки класса можно улучшить: когда класс должен быть загружен, пакет, к которому он принадлежит, идентифицируется и используется для определения конкретного модульного JAR. Только данный JAR сканируется для поиска класса (см. рис. 6.1). Это основывается на предположении, что никакие два модульных JAR не содержат типы в одном и том же пакете — если они так делают, то это называется *разделенным пакетом*, а система модулей выдаст ошибку, как объясняется в разделе 7.2.

Новый тип `AppClassLoader` и его в равной степени новый супертип `BuiltinClassLoader` реализуют новое поведение, а начиная с Java 9, загрузчик классов приложения представляет собой `AppClassLoader`. Это значит, что случайная последовательность `(URLClassLoader) getClass().GetClassLoader()` больше не будет выполняться успешно. Если вы хотите узнать больше о структуре и отношениях загрузчиков классов в Java 9+, то взгляните на подраздел 12.4.1.

6.2.2. Обходимся без URLClassLoader

Если вы столкнулись с приведением к `URLClassLoader` в проекте, от которого зависите, и нет версии, совместимой с Java 9+ и пригодной для обновления, ничего нельзя будет сделать, кроме одного из перечисленных ниже вариантов:

- объявите о проблеме с проектом или предложите исправление;
- разветвите или исправьте проект локально;
- ожидайте.

Если все-таки нашла коса на камень, то можете переключиться на другую библиотеку или фреймворк, при наличии у них версий, которые отлично работают на Java 9+.

Если ваш собственный код выполняет приведение, то вы можете (и должны) что-то с этим сделать. К сожалению, скорее всего, придется отказаться от функции или двух. Вероятнее всего, вы выполните приведение к `URLClassLoader` для использования его конкретного API, и, хотя в `ClassLoader` были добавлены дополнения, он не может полностью заменить `URLClassLoader`. Тем не менее взгляните на него — вероятно, он может делать то, что нужно.

При необходимости просто увидеть путь класса, с которым было запущено приложение, проверьте системное свойство `java.class.path`. Если вы применили `URLClassLoader` для динамической загрузки предоставленного пользователем кода (например, как часть инфраструктуры плагина), добавив JAR-файлы в пути к классам, то нужно найти новый способ сделать это, поскольку с загрузчиком классов приложения, используемым Java 9 и более поздними версиями, сделать это невозможно.

Вместо этого рассмотрите вариант создания нового загрузчика классов, что дает дополнительное преимущество: можно будет избавиться от новых классов, поскольку их не нужно добавлять в загрузчик классов приложения. Если вы компилируете хотя бы под Java 9, то слои приложения могут подойти еще лучше (см. раздел 12.4).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Вы можете испытать желание исследовать `AppClassLoader` и использовать его возможности, если он делает то, что вам нужно. В общем-то, не надо так делать! Полагаться на `AppClassLoader` некрасиво, поскольку это закрытый внутренний класс и вызывать его необходимо с помощью рефлексии. Опираться на его публичный супертип `BuiltinClassLoader` также не рекомендуется.

Как предполагает имя пакета `jdk.internal.loader`, это *внутренний* API; и, поскольку пакет был добавлен в Java 9, он по умолчанию недоступен, вследствие чего придется использовать `--add-exports` или даже `--add-opens` (подробности см. в разделе 7.1). Это не только усложняет код и процесс сборки, но и приводит к возможным проблемам с совместимостью в будущих обновлениях Java — например, при рефакторинге этих классов. Поэтому не делайте так, если только без этого не обойтись при реализации критически важной функции.

6.2.3. Поиск проблемных приведений

Изучать код для этих приведений очень просто: полнотекстовый поиск по «(URLClassLoader)» должен сработать, и результат будет содержать несколько ложных срабатываний (включая скобки, чтобы найти только приведение). Касательно нахождения их в зависимостях, я не знаю ни одного инструмента, который сделал бы данный процесс удобным. Я догадываюсь, что комбинация волшебства инструмента сборки (для получения всех исходных JAR-файлов ваших зависимостей в одном месте), колдовства командной строки (для получения доступа ко всем их файлам `.java` и *их* содержимому файла) и еще одного полнотекстового поиска может помочь.

6.3. Обновленный макет каталога образов среды выполнения

Структура каталогов JDK и JRE эволюционировала постепенно, и неудивительно, что в течение более 20 лет успела накопить пыль. Конечно, одна из причин, по которой каталоги не были реорганизованы, — обратная совместимость. Что является верным для, кажется, всех деталей каждого из них — некоторый код зависит от их конкретного расположения. Два примера:

- отдельные инструменты, особенно IDE, зависят от точного местоположения `rt.jar` (классы, составляющие ядро времени выполнения Java), `tools.jar` (классы поддержки инструментов и утилит) и `src.zip` (исходный код JDK);
- существует код, который ищет команды Java, такие как `javac`, `jar` или `javadoc`, предполагая, что в запущенной JRE есть соседний каталог `bin`, содержащий их, — и это верно, если JRE является частью установки JDK, поскольку содержит папку `bin` с этими командами и рядом папку `jre`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Затем появилась система модулей и порвала с основными допущениями, которые делали возможными эти два примера выше:

- код JDK теперь является модульным и поэтому должен поставляться в отдельных модулях вместо монолитных JAR, таких как `rt.jar` и `tools.jar`;
- с помощью модульной базы Java-кода и такого инструмента, как `jlink`, образы среды выполнения могут создаваться из любого набора модулей.

Начиная с Java 11, больше не существует отдельного пакета JRE. Для запуска программы требуется либо JDK, либо пакет, созданный с помощью `jlink`.

Когда стало ясно, что в системе модулей произойдут некоторые критические изменения, было принято решение пройти весь путь и полностью реорганизовать структуру каталогов образов среды выполнения. Полученные изменения отображены на рис. 6.2. В целом новый макет стал намного проще:

- один каталог `bin` и никаких дублированных двоичных файлов;
- один каталог `lib`;
- один каталог, `conf`, для хранения всех файлов, предназначенных для конфигурации.

Наиболее непосредственное следствие этих изменений — необходимость обновления инструментов разработки, поскольку старые версии, вероятно, не будут работать с установками JDK версии 9 и более поздних. В зависимости от проекта может иметь смысл поискать в нем код, копающийся в папке JDK/JRE, чтобы найти двоичные файлы, файлы свойств или что-то еще.

URL, который вы получали для системных ресурсов, например, из `ClasLoader::getSystemResource`, также изменился. Раньше он имел следующую форму, где `${`*`путь`*`}` — это нечто наподобие `java/lang/String.class`:

```
jar:file:${путь-к-java}/lib/rt.jar!${путь}
```

Сейчас он выглядит как-то так:

```
jrt:/${модуль}/${путь}
```

Все API JDK, которые создают или используют такие URL, работают с новой схемой, но код, созданный вручную, не JDK-кодом, должен быть обновлен под Java 9+.

Кроме того, методы `Class::getResource*` и `ClassLoader::getResource*` больше не читают внутренние ресурсы JDK. Вместо этого для доступа к внутренним ресурсам модуля используйте `Module::getResourceAsStream` или создайте файловую систему JRT следующим образом:

```
FileSystem fs = FileSystems.getFileSystem(URI.create("jrt:/"));
fs.getPath("java.base", "java/lang/String.class"));
```

Подробности о том, как получить доступ к ресурсам, см. в разделе 5.2.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

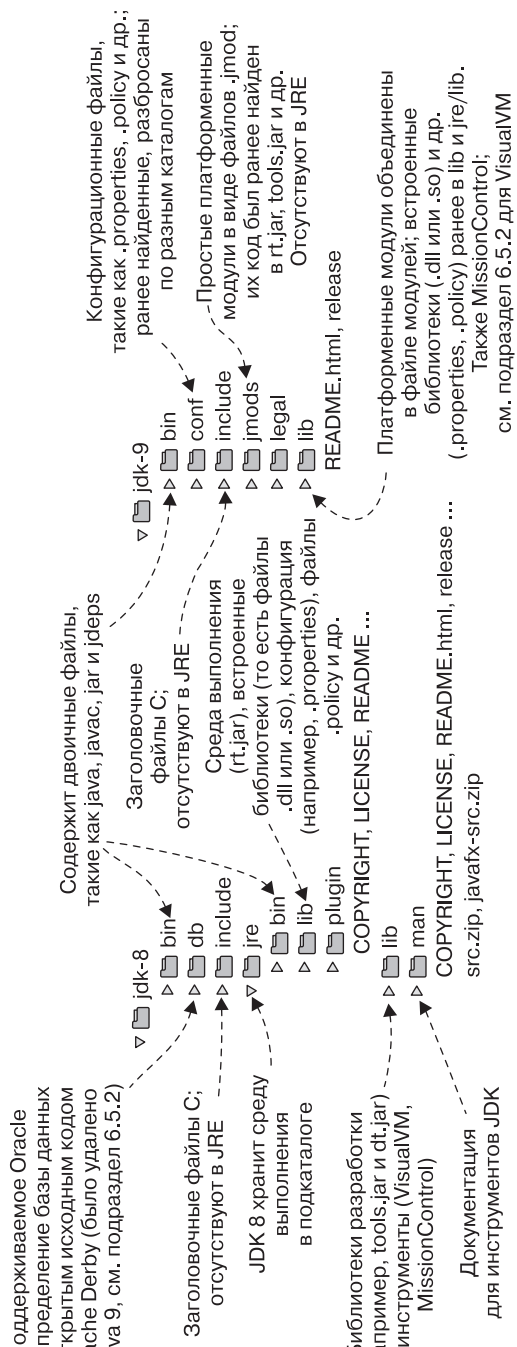


Рис. 6.2. Сравнение структуры каталогов JDK 8 и 9. Новая стала намного стройнее

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

6.4. Выбор, замена и расширение платформы

При компиляции кода или запуске JVM существовали разные способы указать, какие классы составляют платформу JDK. Вы можете выбрать часть JDK, заменить определенную технологию (например, JAXB) другой, добавить несколько классов или выбрать совершенно другую версию платформы для компиляции или запуска. Система модулей сделала часть этих функций устаревшими и переопределила другие с более современным подходом; и независимо от JPMS выпуск Java 9 удалил еще некоторые из них.

Если вы полагаетесь на одну или несколько функций, описанных в этом разделе, то придется приложить некоторые усилия, чтобы проект заработал. Никто не любит, когда его заставляют переделывать что-то, не вызывавшее каких-либо явных проблем. Но, рассматривая эти функции (большинство из которых я никогда не использовал), я могу только представить, насколько упростилось внутреннее устройство JDK без них.

6.4.1. Компактных профилей больше нет

Как объясняется в подразделе 1.5.2, одна из целей системы модулей состояла в том, чтобы позволить пользователям создавать образ среды выполнения только с теми модулями, которые им нужны. Это особенно интересно для небольших устройств с ограниченным объемом памяти и для виртуальных сред, поскольку они заинтересованы в небольших образах среды выполнения. Когда стало очевидно, что система модулей не будет выпущена с Java 8, как планировалось, появилось временное решение — компактные профили.

Три компактных профиля определяют подмножества Java SE 8 API и JRE только с необходимыми классами для поддержки подмножеств этих API. Выбрав профиль, соответствующий требованиям вашего приложения, используйте опцию `javac -profile` для его компиляции (с целью убедиться, что вы остаетесь в выбранном подмножестве), а затем запустите байт-код в соответствующем варианте.

При использовании системы модулей гораздо более гибкие образы среды выполнения можно создать с помощью `jlink` (см. раздел 14.1), и компактные профили больше не понадобятся. Компилятор Java 9+, следовательно, будет принимать `-profile` только при компиляции для Java 8. Для компиляции с определенным выбором модулей можно использовать опцию `--limit-modules`, как описано в подразделе 5.3.5.

Вот список модулей, которые нужны для получения тех же API, что и с тремя компактными профилями:

- для профиля compact1 — *java.base*, *java.logging* и *java.scripting*;
- для профиля compact2 — те же, что и для compact1, плюс *java.rmi*, *java.sql* и *java.xml*;
- для профиля compact3 — те же, что и для compact2, плюс *java.compiler*, *java.instrument*, *java.management*, *java.naming*, *java.prefs*, *java.security.jgss*, *java.security.sasl*, *java.sql.rowset* и *java.xml.crypto*.

Вместо того чтобы полагаться на фиксированный выбор, я рекомендую другой подход. Используйте `jlink` для создания образа только с нужными платформенными модулями (см. раздел 14.1); если приложение и его зависимости полностью модульны, то можно даже добавить модули приложения (см. раздел 14.2).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

6.4.2. Удален механизм расширения

До Java 9 механизм расширения позволял добавлять классы в JDK, не прибегая к необходимости помещать их в путь классов. Он загружал их из разных каталогов: из тех, что названы системным свойством `java.ext.dirs`, из `lib/ext` в JRE, или из системного каталога для конкретной платформы. Java 9 избавляется от этой функции, и компилятор и среда выполнения завершают работу с ошибкой, если находится каталог JRE или задано системное свойство.

Альтернативы следующие:

- опция `java` и `javac --patch-module` внедряет содержимое в модули (см. подраздел 7.2.4).
- опция `java` и `javac --upgrade-module-path` заменяет обновляемый платформенный модуль другим (см. подраздел 6.1.3).
- расширяющиеся артефакты могут быть размещены в пути к классам.

6.4.3. Удален механизм переопределения утвержденных стандартов

До Java 9 механизм переопределения утвержденных стандартов позволял заменять определенные API пользовательскими реализациями. Он загружал их из каталогов, названных системным свойством `java.endorsed.dirs`, или из каталога `lib/endorsed` в JRE. Java 9 удаляет эту функцию, и компилятор и среда выполнения завершают работу с ошибкой, если существует каталог JRE или задано системное свойство. Альтернативы те же, что и для механизма расширения (см. подраздел 6.4.2).

6.4.4. Удалены некоторые параметры пути к загрузочному классу

Параметры `-Xbootclasspath` и `-Xbootclasspath/p` были удалены. Вместо них используйте следующие:

- `javac`-параметр `--system` указывает альтернативный источник системных модулей;
- `javac`-параметр `--release` указывает альтернативную версию платформы;
- `java`- и `javac`-параметр `--patch-module` внедряет содержимое в модули в графе исходных модулей.

6.4.5. Компиляции для Java 5 больше нет

Компилятор Java может обрабатывать источники из различных версий языка Java (например, Java 7 указывается с помощью `-source`), а также генерировать байт-код для различных версий JVM (например, для Java 8 версия указывается с помощью `-target`). Java использовал политику «один вперед, три назад», означающую, что `javac 9` поддерживает Java 9 (очевидно), а также 8, 7 и 6.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Установка `-source 5` или `-target 5` в `javac 8` приводит к предупреждению об устаревании и больше не поддерживается `javac 9`. Аналогично установка `-source 6` или `-target 6` в Java 9 приводит к тому же предупреждению. Теперь, когда релизы выпускаются каждые шесть месяцев, эта политика больше не применяется. Java 10, 11 и 12 могут прекрасно компилироваться для Java 6.

ПРИМЕЧАНИЕ

Компилятор может распознавать и обрабатывать байт-код всех предыдущих JDK — он просто больше не создает байт-код для версий до 6.

6.4.6. Удален выбор версии JRE

До Java 9 можно было использовать опцию `-version:N` в `java` (или соответствующую запись манифеста) для запуска приложения с JRE версии `N`. В Java 9 эта функция была удалена: средство запуска Java завершает работу с ошибкой параметра командной строки и выводит предупреждение для записи манифеста, в противном случае игнорируя его. Если вы полагались на данную функцию, то вот что говорит об этом документация Java:

> Современные приложения обычно развертываются через Java Web Start (JNLP), собственные системы упаковки ОС или активные установщики. Эти технологии имеют собственные методы для управления необходимыми JRE путем поиска, скачивания и обновления требуемых JRE по мере надобности. Вследствие этого выбор версии запуска JRE модуля запуска устаревает.

Похоже, документация считает, что приложения, использующие `-version:N`, несовременны, — что за грубость! Говоря без шуток, если ваше приложение зависит от данной функции, то у вас нет другого выбора, кроме как заставить его работать без `-version:N`, например связывая это с JRE, которая подходит ему больше всего.

6.5. Мелочи, которые приводят к большим неприятностям

Вдобавок к более сложным задачам, создаваемым системой модулей, существует несколько изменений, часто не связанных с JPMS, которые не так значительны, но все равно будут вызывать проблемы:

- новый формат для указания версии;
- удаление ряда инструментов JDK и JRE;
- то, что одинарное подчеркивание больше не является действительным идентификатором;
- обновление синтаксиса протокола запуска сети Java (JNLP);
- удаление параметров JVM.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Я не хочу задерживать вас слишком долго, но и не хочу упускать из виду то, что могло бы остановить миграцию вашего приложения. Поэтому расскажу о каждом изменении, но максимально кратко.

6.5.1. Новый формат версий

Спустя более 20 лет Java окончательно и официально признал, что больше не работает в версии 1.x. Самое время. Отныне системное свойство `java.version` и его производные `java.runtime.version`, `java.vm.version`, `java.specification.version` и `java.vm.specification.version` начинаются не с `1.x`, а с `x`. Аналогично `java -version` возвращает `x`, поэтому в Java 9 вы получите `9.что-то_там`.

Формат версий

Точный формат нового указания версии все еще находится в разработке. В Java 9 указывается `9.${ВТОРОСТЕПЕННАЯ_ВЕРСИЯ}.${БЕЗОПАСНОСТЬ}.${ИСПРАВЛЕНИЕ}`, где `${БЕЗОПАСНОСТЬ}` имеет особенность, заключающуюся в том, что он не сбрасывается в ноль при выпуске новой минорной версии — теперь всегда можно будет узнать, какая версия содержит больше исправлений безопасности, посмотрев на это число.

В Java 10 и более поздних версиях указывается `${ФУНКЦИЯ}.${ПРОМЕЖУТОЧНАЯ_ВЕРСИЯ}.${ОБНОВЛЕНИЕ}.${ИСПРАВЛЕНИЕ}`, где `${ФУНКЦИЯ}` начинается с `10` и увеличивается каждые шесть месяцев с каждым выпуском. `${ПРОМЕЖУТОЧНАЯ_ВЕРСИЯ}`, по идее, должен действовать так же, как и `${ВТОРОСТЕПЕННАЯ_ВЕРСИЯ}`, но, поскольку в новом расписании второстепенные выпуски не запланированы, предполагается, что он всегда останется равным `0`.

К сожалению, побочный эффект заключается в том, что код, отслеживающий версии, может внезапно перестать сообщать о правильных результатах, и это приведет к странному поведению программы. Полнотекстовый поиск соответствующих системных свойств должен найти подобный код.

Касательно обновлений, при необходимости повысить требования проекта до Java 9+ можно отказаться от системных свойств и вместо этого использовать новый тип `Runtime.Version`, что упростит задачу:

```
Version version = Runtime.version();
// в Java 10 и позже используйте `version.feature()`
switch (version.major()) {
    case 9:
        System.out.println("Modularity");
        break;
    case 10:
        System.out.println("Local-Variable Type Inference");
        break;
    case 11:
        System.out.println("Pattern Matching (we hope)");
        break;
}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

6.5.2. Исчезновение инструментов

JDK накопил много инструментов, и со временем часть из них стали излишними или заменились другими. Некоторые были включены в весеннюю чистку Java 9:

- JavaDB больше не включается в сборку. Это была БД Apache Derby, которую можно скачать с db.apache.org;
- VisualVM больше не связана с JDK и стала отдельным проектом на github.com/oracle/visualvm;
- библиотека агента `hprof` была удалена. Инструменты, заменяющие его функции: `jcmd`, `jmap` и Java Flight Recorder;
- визуализатор пространства кучи `jhat` был удален;
- установщики `java-rmi.exe` и `java-rmi.cgi` были удалены. В качестве альтернативы используйте сервлет для прокси RMI через HTTP;
- инструмент `native2ascii` использовался для преобразования пакетов ресурсов на основе UTF-8 в ISO-8859-1. Java 9+ поддерживает комплекты на основе UTF-8, поэтому инструмент стал лишним и был удален.

Кроме того, все связанные с JEE инструменты командной строки, такие как `wsgen` и `xjc`, больше не доступны в Java 11, поскольку были удалены вместе с содержащими их модулями (подробности о модулях JEE см. в разделе 6.1).

6.5.3. Всякие мелочи

Вот, пожалуй, те мелочи, которые могут привести к сбою сборки Java 9: Java 8 сделал устаревшим одинарное подчеркивание _ в качестве идентификатора, а в Java 9 вы получите ошибку компиляции, если будете использовать его как единое целое. Это было сделано, чтобы вернуть подчеркивание как возможное ключевое слово; будущие версии Java придадут ему особое значение.

Другая проблема: `Thread.stop (Throwable)` теперь генерирует исключение `UnsupportedOperationException`. Другие перегрузки `stop` продолжают работать, но их использование крайне не рекомендуется.

Синтаксис JNLP был обновлен в соответствии со спецификацией XML и «для устранения несоответствий, упрощения сопровождения кода и повышения безопасности». Я не буду перечислять изменения — их можно найти по адресу mng.bz/dnfM.

Каждая версия Java удаляет некоторые устаревшие параметры JVM, и Java 9 ничем не отличается в этом плане. Особое внимание уделено сбору мусора, и некоторые комбинации больше не поддерживаются (`DefNew + CMS`, `ParNew + SerialOld`, `Incremental CMS`), а некоторые конфигурации были удалены (`-Xincgc`, `-XX:+CMSIncrementalMode`, `-XX:+UseCMSCompactAtFullCollection`, `-XX:+CMSFullGCsBeforeCompaction`, `-XX:+UseCMSCollectionPassing`) или стали нерекомендуемыми (`-XX:+UseParNewGC`). Java 10, в свою очередь, удаляет `-Xoss`, `-Xsqnopause`, `-Xoptimize`, `-Xboundthreads` и `-Xusealtsigs`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

6.5.4. Новые устаревшие элементы в Java 9, 10 и 11

И наконец, вот неполный список элементов, которые устарели в Java 9, 10 и 11:

- Applet API в пакете `java.applet` вместе с инструментом `appletviewer` и плагином для браузера Java;
- Java Web Start, JNLP и инструмент `javaws`;
- сборщик мусора Concurrent Mark Sweep (CMS);
- HotSpot FlatProfiler, активированный с помощью `-Xprof`;
- инструмент безопасности `policytool`.

Java 10 и 11 продолжили этот список:

- Java 10 удаляет FlatProfiler и `policytool`;
- Java 11 удаляет Applet API и Web Start.

Более подробную информацию, а также детали и предлагаемые альтернативы можно найти в примечаниях к выпуску (Java 9: mng.bz/GLkN; Java 10: mng.bz/zLeV) и в списке устаревшего кода, который помечен для удаления (Java 9: mng.bz/YX9e; Java 10: mng.bz/qRoU).

Резюме

- Модули JEE устарели в Java 9 и удалены в Java 11. Вам нужно найти стороннюю зависимость, которая отвечает всем требованиям, и чем раньше, тем лучше.
- В Java 9 и 10 эти модули по умолчанию не распределены, следовательно, возможны ошибки во время компиляции и выполнения. Чтобы это исправить, используйте стороннюю зависимость, которая реализует тот же API, или сделайте модуль JEE доступным с помощью `--add-modules`.
- Загрузчик классов приложения больше не относится к типу `URLClassLoader`, поэтому код, подобный `(URLClassLoader) getClass().GetClassLoader()`, завершится с ошибкой. Возможные решения: полагаться только на `ClassLoader` API, даже если это значит, что функция должна быть удалена (рекомендуется); создать слой для динамической загрузки нового кода (рекомендуется); или взломать внутренние компоненты загрузчика классов и использовать `BuiltinClassLoader` или даже `AppClassLoader` (не рекомендуется).
- Структура каталогов образа среды выполнения изменилась, и вам, вероятно, придется обновить инструменты, в частности IDE, для работы с Java 9 и более поздними версиями. Кроме того, необходимо обновить код в каталогах JDK/JRE или вручную созданных URL для системных ресурсов.
- Несколько механизмов, которые модифицировали набор классов, составляющих платформу, были удалены. Для большинства из них система модулей предлагает альтернативы:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- вместо использования компактных профилей создайте образы среды выполнения с помощью `jlink` и настройте компиляцию, прибегнув к `--limit-modules`;
- вместо механизма расширения или механизма утвержденных стандартов примените `--patch-module`, `--upgrade-module-path` или путь класса;
- вместо опции `-Xbootclasspath` задействуйте `--system`, `--release` или `--patch-module`.

- Теперь нельзя компилировать под Java 5 или использовать параметр `-version:N` для запуска приложения с Java версии `N`.
- Инструменты командной строки Java и системное свойство `java.version` сообщают о своей версии как `9.${ВТОРОСТЕПЕННАЯ_ВЕРСИЯ}.${`*`БЕЗОПАСНОСТЬ`*`}.${`*`ИСПРАВЛЕНИЕ`*`}` (в Java 9) или как `${ФУНКЦИЯ}.${ПРОМЕЖУТОЧНАЯ_ВЕРСИЯ}.${ОБНОВЛЕНИЕ}.${`*`ИСПРАВЛЕНИЕ`*`}` (в Java 10 и более поздних версиях), то есть в Java X они начинаются с `x` вместо `1.x`. Новый API `Runtime.Version` делает анализ этого свойства ненужным.
- Следующие инструменты были удалены:
  - в Java 9: JavaDB, VisualVM, `hprof`, `jhat`, `java-rmi.exe`, `java-rmi.cgi` и `native2ascii`;
  - в Java 10: `policytool`;
  - в Java 11: `idlj`, `orbd`, `schemagen`, `servertool`, `tnameserv`, `wsgen`, `wsimport` и `xjc`.
- Одинарное подчеркивание больше не является правильным идентификатором.
- Синтаксис JNLP был обновлен в соответствии со спецификацией XML, поэтому, возможно, придется обновить файлы JNLP.
- Каждая версия Java удаляет устаревшие параметры командной строки JVM, что может привести к поломке некоторых существующих сценариев.
- Java 9 не поддерживает технологию Applet и Java Web Start, а Java 11 удаляет их.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

7 Повторяющиеся проблемы при переходе на Java 9 и выше

В этой главе

- Различия в стандартизированных, поддерживаемых и внутренних API JDK.
- Поиск зависимостей от внутренних API JDK с помощью JDeps.
- Компиляция и запуск кода, зависящего от внутренних API.
- Причины, по которым разделенный пакет может сделать классы невидимыми.
- Исправление разделенных пакетов.

В главе 6 обсуждается ряд проблем, с которыми вы можете столкнуться при переносе проекта на Java 9+. Однако сразу по окончании этого вы больше не столкнетесь с ними, если только не выберете зависимости до Java 9. В текущей главе рассматриваются две проблемы, которые, возможно, придется как-то решать:

- использование внутренних API приводит к ошибкам компиляции (см. раздел 7.1). Это верно не только для JDK-внутренних API, таких как классы из пакетов `sun.*`, но и для кода, внутреннего для библиотек или сред, от которых вы зависите;
- разделение пакетов по артефактам приводит к ошибкам во время компиляции и во время выполнения (см. раздел 7.2). Опять же это может случиться между вашим кодом и модулями JDK, а также между любыми другими двумя артефактами: например, между вашим кодом и сторонней зависимостью.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Точно так же, как и с проблемами, которые обсуждались до сих пор, вам придется работать над этими двумя проблемами, когда проект будет запущен на Java 9+. Однако на этом все не заканчивается: можно иногда столкнуться с ними даже после миграции, при работе с кодом или вытягивании новых зависимостей. Зависимости от внутренних модулей и разделенных пакетов вызывают проблемы, невзирая на типы используемых модулей. С такой же вероятностью вы столкнетесь с ними в коде пути класса и в платформенных модулях (сценарий перехода), как и в модулях приложения (сценарий, в котором вы уже работаете на Java 9 или более поздней версии и применяете модули).

В данной главе показано, как может сломаться инкапсуляция модуля и как исправить разделение пакетов, независимо от контекста, в котором возникают эти ситуации. В совокупности с материалами главы 6 это подготовит вас к большинству сценариев, которые могут пойти не так во время миграции.

О пути класса

Если вы не читали примечание в главе 6, то я приведу его здесь.

- Путь класса все еще полностью функционален. Во время перехода на Java 9+ вы продолжите использовать его вместо пути модуля.
- Даже в этом случае система модулей все еще работает: например, в случаях разрешения модулей.
- Код в пути к классам автоматически считывает большинство модулей (но не все, см. раздел 6.1), поэтому они доступны во время компиляции или выполнения без дополнительной настройки.

7.1. Инкапсуляция внутренних API

Одно из главных преимуществ системы модулей — строгая инкапсуляция. Как подробно объясняется в разделе 3.3, мы наконец можем убедиться, что лишь поддерживаемые API доступны для внешнего кода, сохраняя при этом детали реализации скрытыми.

Недоступность внутренних API применяется к платформенным модулям, поставляемым с JDK, где полностью поддерживаются только пакеты `java.*` и `javax.*`. Например, это происходит, когда вы пытаетесь скомпилировать класс со статической зависимостью (имеется в виду импорт или использование полного имени класса, а не рефлексивный доступ) для `NimbusLookAndFeel` в инкапсулированном пакете `com.sun.java.swing.plaf.nimbus`:

```
> error: package com.sun.java.swing.plaf.nimbus is not visible
> import com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel;
>                                     ^
>     (package com.sun.java.swing.plaf.nimbus is declared
>      in module java.desktop, which does not export it)
> 1 error
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Удивительно, что многие библиотеки и фреймворки, а также код приложения (часто важнейшие его части) используют классы из пакетов `sun.*` или `com.sun.*`, большинство из которых недоступны в Java 9 и выше. В этом разделе я покажу, как найти подобные зависимости и что с ними делать.

Но для чего это нужно обсуждать? Если внутренние API недоступны, то говорить не о чем, верно? Что ж, пришло время рассказать кое о чем: они не совсем недоступны. Во время выполнения все будет работать до следующего основного выпуска Java (хотя могут появиться некоторые нежелательные предупреждающие сообщения); и при наличии контроля над командной строкой любой пакет можно сделать доступным во время компиляции. (Мне кажется, я только что услышал вздох облегчения, — это были вы?)

В подразделе 9.1.4 обсуждаются более широкие последствия использования параметров командной строки для настройки системы модулей; здесь мы сосредоточены на решении насущной проблемы. Мы будем различать статический и рефлексивный доступ, а также доступ во время компиляции и во время выполнения (см. подразделы 7.1.3 и 7.1.4), поскольку между ними есть некоторые критические различия. Но прежде, чем переходить к этому, нужно иметь точное представление, что же такое внутренний API и как инструмент анализа зависимостей Java (JDeps) способен помочь найти проблемный код в проекте и зависимостях.

ПРИМЕЧАНИЕ

> Если вы не уверены, как именно работает рефлексия, то загляните в приложение Б, в котором дается краткий обзор. Кроме того, в этом разделе мы сосредоточимся на рефлексивном доступе в JDK; для более общего взгляда на рефлексию в модульном мире см. главу 12.

По окончании работы с этим разделом вам будет легко взломать открытые модули, чтобы применить API, которые не должны были использоваться (по мнению их разработчиков). И еще более важна возможность оценить преимущества и недостатки данной стратегии, чтобы принять обоснованное решение о том, стоит ли идти по этому пути.

7.1.1. Внутренние API под микроскопом

Какие API — внутренние? В целом это каждый класс, который не является общедоступным или отсутствует в экспортируемом пакете. Данное правило полностью применяется и к модулям приложения. Что касается JDK, то ответ не так прост. Помимо уже исторически сложной ситуации со стандартизированными, поддерживаемыми и внутренними API, Java 9+ добавляет уровень сложности, создавая особые случаи для одних API и удаляя другие. Разберем ситуацию шаг за шагом.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Три вида API JDK: стандартизированные, поддерживаемые и внутренние

Java Runtime Environment (JRE) поддерживает три вида API.

- Публичные классы в пакетах `java.*` и `javax.*` стандартизированы и полностью поддерживаются во всех JRE. Использование только их создает наиболее переносимый код.
- Некоторые пакеты `com.sun.*` и `jdk.*` и содержащиеся в них классы помечаются аннотацией `jdk.Exported`, и в этом случае они поддерживаются Oracle, но не обязательно присутствуют в JRE не от Oracle. Эта зависимость связывает код с конкретными JRE.
- Большинство пакетов `com.sun.*` и все пакеты `sun.*`, а также все непубличные классы являются внутренними и могут изменяться между различными версиями и JRE. Зависимость от них наиболее нестабильна, поскольку такой код теоретически может перестать работать при любом незначительном обновлении.

На момент появления Java 9+ и системы модулей все эти три вида API — стандартизированный, поддерживаемый и внутренний — все еще существуют. Ключевое значение имеет то, является ли модуль экспортом пакета, но, очевидно, трех выделенных категорий недостаточно. Еще один индикатор — имя модуля. Как вы помните из подраздела 3.1.4, платформенные модули делятся на модули, определенные в спецификации Java (с префиксом *java.**) и специфические для JDK (с префиксом *jdk.**):

- публичные классы, найденные в пакетах, экспортируемых модулями *java.** (это могут быть пакеты `java.\*` и `javax.*`), *стандартизированы*;
- публичные классы, найденные в пакетах, экспортируемых модулями *jdk.**, не стандартизированы, но *поддерживаются* в JDK Oracle и OpenJDK;
- все остальные классы являются внутренними API.

То, какие конкретные классы стандартизированы, поддерживаются или являются внутренними, осталось в значительной степени неизменным от Java 8 до Java 9+. Как следствие, многие классы в `com.sun.*` и все классы в `sun.*` — внутренние API, как и прежде. Разница в том, что система модулей превращает это соглашение в активно применяемое отличие. Рисунок 7.1 показывает разделение, в котором внутренние API не экспортируются.

То, что модули *jdk.** не стандартизированы, — всего лишь соглашение, и система модулей о том не знает. Поэтому, хотя может быть неразумно зависеть от их экспортируемых API, JPMS не будет их инкапсулировать, и ни один из параметров командной строки, которые мы обсудим, не обязателен. Здесь, говоря о внутренних API, я имею в виду те, которые система модулей делает недоступными, поскольку классы не являются общедоступными или пакеты не экспортируются.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

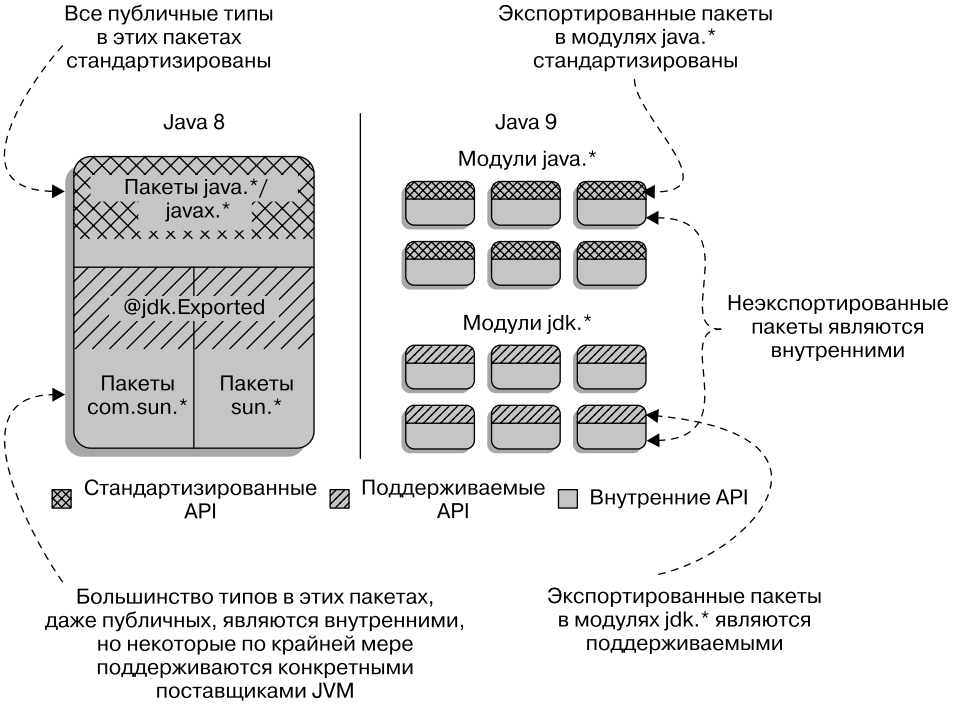


Рис. 7.1. В Java 8 *(слева)* имена пакетов и редко встречающаяся аннотация @jdk.Exported определяли, был ли API стандартизированным, поддерживаемым или внутренним. Начиная с Java 9 *(справа)*, эту функцию выполняют имена модулей и директивы экспорта

Особый случай для печально известного sun.misc.Unsafe

Как вы могли догадаться, первоначальная идея заключалась в том, чтобы инкапсулировать каждый API, который был внутренним до Java 9. Это вызвало переполох, когда большая часть сообщества Java осознала это в 2015 году. Хотя средний Java-разработчик может лишь иногда применять внутренние API, многие из наиболее известных библиотек и фреймворков часто ими пользуются и от них зависит часть их наиболее важных функций.

Живой пример этой ситуации — `sun.misc.Unsafe`, класс, с учетом имени пакета явно внутренний. Он предлагает функциональность, необычную для Java и, как следует из названия класса, небезопасную. (Поговорим о выразительных именах!) Возможно, лучший пример — прямой доступ к памяти, который JDK должен иногда выполнять.

Но это выходило за рамки JDK. Благодаря доступности `Unsafe` некоторые библиотеки, особенно ориентированные на высокую производительность, начали использовать его; со временем большие части экосистемы оказались в прямой или косвенной зависимости от него. Перспектива того, что этот класс и другие, подобные ему, будут инкапсулированы, привела к шумихе внутри сообщества.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

После этого команда, работающая над Project Jigsaw, решила предоставить более плавный путь миграции. Исследование существующих внутренних API и их использования вне JDK дало следующий результат.

- Большинство затронутых API редко или никогда не используются.
- Некоторые уязвимые API иногда применяются, но стандартизированные альтернативы существовали и до Java 9. Яркий пример — пара `BASE64Encoder/BASE64Decoder` в `sun.misc`, которую можно заменить на `java.util.Base64`.
- Те или иные уязвимые API используются время от времени, но предоставляют критически важную функциональность, для которой нет альтернативы. В том числе и `sun.misc.Unsafe`.

Было принято решение инкапсулировать первые два вида, но оставить третий доступным как минимум для другой основной версии Java. Однако их экспорт из соответствующих модулей мог бы сбить с толку, поскольку сделал бы их похожими на поддерживаемые или даже стандартизированные API, что, скорее всего, не так. Нет лучшего способа указать на это, чем создать модуль с соответствующим названием.

Критические API, для которых до Java 9 не было замен, экспортируются модулем *jdk.unsupported*. Как следует из названия, он специфичен для JDK (гарантированно присутствует только в Oracle JDK и OpenJDK) и не поддерживается (содержимое может измениться в следующем выпуске). В Java 9–11 он содержит такие классы:

- из `sun.misc`: `Signal`, `SignalHandler` и `Unsafe`;
- из `sun.reflect`: `Reflection` и `ReflectionFactory`;
- из `com.sun.nio.file`: `ExtendedCopyOption`, `ExtendedOpenOption`, `ExtendedWatchEventModifier` и `SensitivityWatchEventModifier`.

Если ваш код или зависимости зависят от данных классов (подраздел 7.1.2 показывает, как это выяснить), то, хотя они были внутренним API до Java 9, не нужно ничего делать, чтобы продолжать их использовать. Но это пока. После выпуска стандартизированных альтернатив их функциональности (например, переменных дескрипторов, которые заменяют части `Unsafe`) они будут инкапсулированы. Я настоятельно рекомендую внимательно изучить применение этих классов и подготовиться к их возможному исчезновению.

Удаленные API

Хотя некоторые внутренние API останутся доступными еще несколько лет и большинство из них были инкапсулированы, часть из них подверглась беспощадному удалению или переименованию. Это нарушает код, который использует их вне досягаемости любого переходного периода и параметра командной строки. Вот они:

- все в `sun.misc` и `sun.reflect`, что не является частью *jdk.unsupported*: например, `sun.misc.BASE64Encoder`, `sun.misc.BASE64Decoder`, `sun.misc.Cleaner` и `sun.misc.Service`;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- `com.sun.image.codec.jpeg` и `sun.awt.image.codec`;
- `com.apple.concurrent`;
- `com.sun.security.auth.callback.DialogCallbackHandler`;
- методы `addPropertyChangeListener` и `removePropertyChangeListener` для `java.util.logging.LogManager`, `java.util.jar.Pack200.Packer` и `java.util.jar.Pack200.Unpacker` (устарели в Java 8);
- методы с параметрами или типами возврата из `java.awt.peer` и `java.awt.dnd.peer` (эти пакеты никогда не были стандартизированы и являются внутренними в Java 9 и более поздних версиях).

Большинство этих классов и пакетов имеют альтернативы, и можно использовать JDeps, чтобы узнать о них.

7.1.2. Анализ зависимостей с помощью JDeps

Теперь, когда мы обсудили различие между стандартизированными, поддерживаемыми и внутренними API и особым случаем *jdk.unsupported*, пришло время применить данные знания в реальном проекте. Чтобы обеспечить его совместимость с Java 9+, вам нужно выяснить, от каких внутренних API это зависит.

Простое ознакомление с кодовой базой проекта не изменит его — возникнут проблемы, если библиотеки и фреймворки зависят от причинных проблем, вследствие чего нужно проанализировать их. Это звучит как ужасная ручная работа, отсеивание глыб кода в поисках ссылок на такие API. К счастью, в этом нет необходимости.

Начиная с Java 8, JDK поставляется со встроенным в командную строку *Java-инструментом анализа зависимостей* (JDeps). Он анализирует байт-код Java, то есть файлы `.class` и JAR, и записывает все статически объявленные зависимости между классами, которые затем могут быть отфильтрованы или агрегированы. Это отличный инструмент для визуализации и изучения различных графов зависимостей, о которых я говорил. Приложение Г содержит учебник по JDeps; вы можете прочитать его, если до этого никогда не использовали данный инструмент. Однако понимать этот раздел не обязательно.

Одна особенность наиболее интересна в контексте внутренних API: параметр `--jdk-internals` заставляет JDeps перечислять все внутренние API, от которых зависят JAR, на которые присутствуют ссылки, в том числе экспортируемые *jdk.unsupported*. Вывод содержит следующее:

- анализируемый JAR и модуль, содержащий проблемный API;
- конкретные участвующие классы;
- причину, по которой зависимость проблематична.

Я собираюсь использовать JDeps в Scaffold Hunter, «инструменте с открытым исходным кодом на основе Java для визуального анализа наборов данных». Следующая команда анализирует внутренние зависимости:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
$ jdeps --jdk-internals
    -R --class-path 'libs/*'
    scaffold-hunter-2.6.3.jar
```

Дает JDeps указание проанализировать использование внутренних API

Рекурсивно анализирует все зависимости

Начинает с JAR-файлов приложения

Вывод начинается с упоминания о разделенных пакетах, которые мы рассмотрим в разделе 7.2. Затем он сообщает о проблемных зависимостях, часть которых показана далее. Вывод подробный и дает всю информацию, необходимую для изучения рассматриваемого кода или открытых вопросов в соответствующих проектах:

```
> batik-codec.jar -> JDK removed internal API
>     JPEGImageWriter -> com.sun.image.codec.jpeg.JPEGCodec
>         JDK internal API (JDK removed internal API)
>     JPEGImageWriter -> com.sun.image.codec.jpeg.JPEGEncodeParam
>         JDK internal API (JDK removed internal API)
>     JPEGImageWriter -> com.sun.image.codec.jpeg.JPEGImageEncoder
>         JDK internal API (JDK removed internal API)
# [...]
> guava-18.0.jar -> jdk.unsupported
>     Striped64 -> sun.misc.Unsafe
>         JDK internal API (jdk.unsupported)
>     Striped64$1 -> sun.misc.Unsafe
>         JDK internal API (jdk.unsupported)
>     Striped64$Cell -> sun.misc.Unsafe
>         JDK internal API (jdk.unsupported)
# [...]
> scaffold-hunter-2.6.3.jar -> java.desktop
>     SteppedComboBox -> com.sun.java.swing.plaf.windows.WindowsComboBoxUI
>         JDK internal API (java.desktop)
>     SteppedComboBox$1 -> com.sun.java.swing.plaf.windows.WindowsComboBoxUI
>         JDK internal API (java.desktop)
```

JPEGImageWriter (я сократил пакет) зависит от нескольких разных классов

batik-codec зависит от удаленного API

Определяет, в чем заключается проблема

Guava зависит от jdk.unsupported

Striped64 зависит от sun.misc.Unsafe, и то же относится к двум из его внутренних классов

Scaffold Hunter зависит от классов, внутренних для java.desktop

JDeps заканчивается следующей заметкой, содержащей полезную справочную информацию и предложения по некоторым из обнаруженных проблем:

```
> Warning: JDK internal APIs are unsupported and private to JDK
> implementation that are subject to be removed or changed incompatibly
> and could break your application. Please modify your code to eliminate
> dependence on any JDK internal APIs. For the most recent update on JDK
> internal API replacements, please check:
> https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool
>
> JDK Internal API                          Suggested Replacement
> ----------------                          ---------------------
> com.sun.image.codec.jpeg.JPEGCodec        Use javax.imageio @since 1.4
> com.sun.image.codec.jpeg.JPEGDecodeParam  Use javax.imageio @since 1.4
> com.sun.image.codec.jpeg.JPEGEncodeParam  Use javax.imageio @since 1.4
> com.sun.image.codec.jpeg.JPEGImageDecoder Use javax.imageio @since 1.4
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
> com.sun.image.codec.jpeg.JPEGImageEncoder  Use javax.imageio @since 1.4
> com.sun.image.codec.jpeg.JPEGQTable        Use javax.imageio @since 1.4
> com.sun.image.codec.jpeg.TruncatedFileException
>                                            Use javax.imageio @since 1.4
> sun.misc.Unsafe                            See JEP 260
> sun.reflect.ReflectionFactory              See JEP 260
```

7.1.3. Компиляция с помощью внутренних API

Цель строгой инкапсуляции состоит в том, что система модулей по умолчанию не позволяет использовать внутренние API. Это влияет на компиляцию *и* поведение во время выполнения любой версии Java, начиная с 9. Здесь мы обсуждаем компиляцию — подраздел 7.1.4 посвящен поведению во время выполнения. Вначале сильная инкапсуляция будет в основном иметь отношение к платформенным модулям, но, поскольку ваши зависимости являются модульными, вы увидите тот же барьер вокруг их кода.

Однако иногда можно оказаться в ситуации, когда абсолютно *необходимо* использовать открытый класс в неэкспортированном пакете для решения проблемы. К счастью, это возможно даже при установленной системе модулей. (Я утверждаю очевидное, но хочу отметить: эта проблема относится только к коду, поскольку зависимости уже скомпилированы — они все равно будут подвержены сильной инкапсуляции, но только во время выполнения.)

Экспортирование в модуль

Параметр `--add-exports ${`*`модуль`*`}/${`*`пакет`*`}=${`*`читаемый_модуль`*`}`, доступный для команд `java` и `javac`, экспортирует `${`*`пакет`*`}` из `${`*`модуль`*`}` в `${`*`читаемый_модуль`*`}`. Таким образом, код в `${`*`читаемый_модуль`*`}` может обращаться ко всем открытым типам в `${`*`пакет`*`}`, но другие модули не могут этого делать.

При установке *`${читаемый_модуль}`* в `ALL-UNNAMED` весь код из пути класса может получить доступ к этому пакету. При переходе на Java 9+ вы всегда будете использовать этот заместитель — только после запуска собственного кода в модулях можно будет ограничить экспорт конкретными модулями.

До сих пор экспорт всегда был нецелевым, поэтому возможность экспорта в определенные модули становится новым аспектом. Данная функция доступна и для дескрипторов модулей, как объясняется в разделе 11.3. Я также немного волнуюсь из-за того, что означает `ALL-UNNAMED`. Он связан с безымянным модулем, который подробно рассматривается в разделе 8.2, но сейчас «весь код из пути к классам» можно считать хорошим допущением.

Вернемся к коду, который вызвал следующую ошибку компиляции:

```
> error: package com.sun.java.swing.plaf.nimbus is not visible
> import com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel;
>
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
>     (package com.sun.java.swing.plaf.nimbus is declared
>      in module java.desktop, which does not export it)
> 1 error
```

Здесь какой-то класс (который я исключил из вывода, поскольку он не имеет значения) импортирует `NimbusLookAndFeel` из инкапсулированного пакета `com.sun.java.swing.plaf.nimbus`. Обратите внимание, как сообщение об ошибке указывает на конкретную проблему, включая модуль, содержащий класс.

Это явно так не работает в чистом Java 9, но что, если вы хотите продолжать использовать его? Тогда вы, вероятно, ошибетесь, поскольку в `javax.swing.plaf.nimbus` есть стандартная альтернатива; в Java 10 остается только данная версия ввиду удаления внутренней версии. Но ради этого примера допустим, вы все еще хотите применить внутреннюю версию — возможно, для взаимодействия с устаревшим кодом, который нельзя изменить.

Все, что нужно сделать для успешной компиляции с `com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel`, — это добавить `--add-exports java.desktop/com.sun.java.swing.plaf.nimbus=ALL-UNNAMED` в команду компилятора. Если сделать это вручную, то получится примерно следующее (все заместители должны быть заменены конкретными значениями):

```
$ javac
    --add-exports java.desktop/com.sun.java.swing.plaf.nimbus=ALL-UNNAMED
    --class-path ${dependencies}
    -d ${target-folder}
    ${исходные_файлы}
```

С помощью инструмента сборки нужно будет поместить параметр где-либо в дескрипторе сборки. Просмотрите документацию используемого инструмента, чтобы узнать, как добавить параметры командной строки в компилятор.

Таким образом, код успешно компилируется с инкапсулированными классами. Но важно понимать, что вы только передвинули проблему во время выполнения! Добавление этого экспорта в командной строке изменяет лишь компиляцию — в полученный байт-код не помещается информация, которая позволила бы данному классу получить доступ к пакету во время выполнения. Все еще нужно выяснить, как заставить результат работать во время выполнения.

7.1.4. Выполнение с помощью внутренних API

Я упоминал, что по крайней мере в Java 9, 10 и 11 внутренние JDK-зависимости все еще доступны во время выполнения. Учитывая все вышесказанное, это должно немного вас удивить. На протяжении всей книги я рассказывал о преимуществах сильной инкапсуляции и говорил, что она так же важна, как модификаторы видимости, — почему же она не применяется во время выполнения?

Как и многие другие Java-причуды, эта была рождена из-за приверженности к обратной совместимости: сильная инкапсуляция внутренних компонентов JDK разрушит множество приложений. Даже если это просто устаревшее применение Nimbus, приложение будет аварийно завершено. Сколько пользователей или ИТ-отделов

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

установят Java 9+, если устаревшие приложения перестанут работать? Сколько команд разрабатывали бы что-либо на Java 9+, пользуйся им единицы?

С целью убедиться, что система модулей не разделяет экосистему на «до Java 9» и «после Java 9», было принято решение предоставить коду в пути к классам нелегальный доступ к внутренним API JDK, по крайней мере до Java 11. Каждый из этих аспектов был выбран сознательно:

- *к коду в пути к классам...* — запуск кода из пути модуля выражает, что тот был подготовлен для системы модулей, и в этом случае нет необходимости делать исключение. Следовательно, он ограничен кодом пути класса;
- *...к внутренним* API JDK — с точки зрения совместимости нет причин предоставлять доступ к модулям приложения, поскольку они не существовали до Java 9. Поэтому исключение ограничено платформенными модулями;
- *...по крайней мере* в Java 11 — если бы исключение было постоянным, то желания обновлять проблемный код было бы намного меньше.

Как вы заметили в главе 6, это не решает всех проблем, с которыми может столкнуться приложение при выполнении на Java 9, 10 или 11, но с большей вероятностью его удастся успешно запустить.

Управление полным нелегальным доступом к внутренним API JDK

Для успешной миграции важно понять детали скрытого нелегального доступа к внутренним API JDK; но их изучение сделает вашу мысленную модель системы модулей более сложной. Это помогает усвоить общую картину: строгая инкапсуляция запрещает доступ ко всем внутренним API во время компиляции и выполнения. Вдобавок ко всему, было создано большое исключение, особый дизайн которого обусловился проблемами совместимости. Однако со временем оно исчезнет и мы вернемся к гораздо более четкому поведению.

При разрешении коду в пути к классам обращаться к внутренним API JDK делается различие между кодом, статически зависимым от них, и кодом, который обращается к ним рефлексивно.

- *Рефлексивный доступ вызывает предупреждения*. Поскольку невозможно четко идентифицировать все такие вызовы с помощью статического анализа, точно сообщить о них можно только во время выполнения.
- *Статический доступ не вызывает предупреждения*. Его можно легко обнаружить во время компиляции или с помощью JDeps. Из-за вездесущести статического доступа он также является областью, чувствительной к производительности, где проблематична проверка и иногда отправка сообщений журнала.

Точное поведение можно настроить с помощью параметра командной строки. Параметр `java --illegal-access=${`*`значение`*`}` управляет обработкой нелегального доступа к внутренним API JDK, где `${`*`значение`*`}` представляет собой что-либо из перечисленного ниже:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- `permit` — доступ ко всем внутренним API JDK разрешен для кода в пути к классам. В случае рефлексивного доступа выдается одно предупреждение для первого доступа к каждому пакету;
- `warn` — ведет себя как `permit`, но выдается предупреждение при каждом рефлексивном доступе;
- `debug` — ведет себя как `warn`, но в каждое предупреждение включается трассировка стека;
- `deny` — вариант для тех, кто верит в строгую инкапсуляцию: любой нелегальный доступ по умолчанию запрещен.

В Java 9–11 `permit` — значение по умолчанию. В некоторых будущих версиях Java `deny` станет значением по умолчанию; и в какой-то момент весь параметр может исчезнуть, но я уверен, что это займет еще несколько лет.

Похоже, что после того, как вы пропустите проблемный код через компилятор либо с помощью версии Java 8, либо путем добавления необходимых параметров в версию Java 9+, среда выполнения Java 9+ будет запускать его не слишком охотно. Чтобы увидеть `--illegal-access` в действии, пришло время наконец взглянуть на класс, который играет с внутренним видом и поведением Nimbus:

```
import com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel;

public class Nimbus {

    public static void main(String[] args) throws Exception {
        NimbusLookAndFeel nimbus = new NimbusLookAndFeel();
        System.out.println("Static access to " + nimbus);

        Object nimbusByReflection = Class
                .forName("com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel")
                .getConstructor()
                .newInstance();
        System.out.println("Reflective access to " + nimbusByReflection);
    }

}
```

Он не делает ничего особенно полезного, однако явно пытается получить статический и рефлексивный доступ к `NimbusLookAndFeel`. Чтобы скомпилировать этот код, нужно использовать `--add-exports`, как описано выше. Запустить его проще:

```
$ java --class-path ${class} j9ms.internal.Nimbus

> Static access to "Nimbus Look and Feel"
> WARNING: An illegal reflective access operation has occurred
> WARNING: Illegal reflective access by j9ms.internal.Nimbus
>     (file:...) to constructor NimbusLookAndFeel()
> WARNING: Please consider reporting this to the maintainers
>     of j9ms.internal.Nimbus
> WARNING: Use --illegal-access=warn to enable warnings of
>     further illegal reflective access operations
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
> WARNING: All illegal access operations will be denied in a
>     future release
> Reflective access to "Nimbus Look and Feel"
```

Вы можете наблюдать поведение, определенное параметром по умолчанию `--illegal-access=permit`: статический доступ выполняется без комментариев, но рефлексивный приводит к длинному предупреждению. Установка параметра `warn` ничего не изменит, поскольку есть только один доступ, а `debug` добавляет трассировку стека для проблемного вызова. С помощью `deny` вы получите те же сообщения, которые появлялись в подразделе 3.3.3 при тестировании требований доступности:

```
$ java
    --class-path ${класс}
    --illegal-access=deny
    j9ms.internal.Nimbus

> Exception in thread "main" java.lang.IllegalAccessError:
>     class j9ms.internal.Nimbus (in unnamed module @0x6bc168e5) cannot
>     access class com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel (in
>     module java.desktop) because module java.desktop does not export
>     com.sun.java.swing.plaf.nimbus to unnamed module @0x6bc168e5
```

Есть еще одна деталь для обсуждения: что происходит с нелегальным доступом к внутренним компонентам JDK, представленным в Java 9? Поскольку параметр `--illegal-access` был введен для упрощения миграции, было бы обидно, если бы он *усложнил* возможный переход, прибавив несколько лет к началу работы в зависимости от *новых* внутренних API. Это действительно рискованно!

ВАЖНАЯ ИНФОРМАЦИЯ

Чтобы минимизировать риск зависимости от новых внутренних API JDK, --illegal-access не применяется к пакетам, представленным в Java 9. Это сокращает набор новых API, от которых проекты могут случайно зависеть, до классов, добавленных в пакеты, существовавшие до Java 9.

Все, что было сделано для совместимости, как я и говорил, усложнится еще больше. И я еще не закончил, поскольку мы можем более конкретно управлять нелегальным доступом (см. следующий пункт). Таблица 7.1, представленная ниже, в подразделе 7.1.5, сравнит различные варианты.

Управление специфическим нелегальным доступом к выбранным API

Параметр `illegal-access` характеризуется тремя основными свойствами:

- он управляет незаконным доступом в манере «все и сразу»;
- это переходный вариант, который в конечном итоге исчезнет;
- он раздражает постоянными предупреждениями.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Что произойдет, когда он исчезнет? Станет ли строгая инкапсуляция непреодолимой? Ответ: нет, не станет. Всегда будут крайние случаи, которые требуют доступа к внутренним API (модулей платформы и приложения), и, следовательно, должен существовать некий механизм (возможно, не слишком удобный), чтобы сделать это возможным. И снова мы переходим к параметрам командной строки.

ВАЖНАЯ ИНФОРМАЦИЯ

Как я упоминал в подразделе 7.1.3 при обсуждении внутренних API во время компиляции, --add-exports доступен и для команды java. Он работает точно так же и делает указанный пакет доступным либо для указанного модуля, либо для всего запущенного кода. Это значит, что такой код могут использовать в этих пакетах публичные члены публичных типов, которые охватывают весь статический доступ.

Класс `NimbusLookAndFeel` — публичный, поэтому для правильного доступа нужно лишь экспортировать пакет, который его содержит. Чтобы убедиться в срабатывании `--add-exports`, деактивируйте разрешение по умолчанию для нелегального доступа с помощью `--illegal-access=deny`:

```
$ java
    --class-path ${класс}
    --illegal-access=deny
    --add-exports java.desktop/com.sun.java.swing.plaf.nimbus=ALL-UNNAMED
    j9ms.internal.Nimbus

> Static access to ${вид и поведение Nimbus}
> Reflective access to ${вид и поведение Nimbus}
```

Рефлексивный доступ выполнится. Обратите также внимание, что предупреждения не будет — об этом чуть позже.

Это охватывает доступ к публичным членам публичных типов, но рефлексия способна на большее: благодаря щедрому использованию `setAccessible(true)` она позволяет взаимодействовать с непубличными классами, а также с непубличными полями, конструкторами и методами. Даже в экспортированном пакете данные члены инкапсулированы, поэтому, чтобы успешно применить рефлексию, потребуются дополнительные усилия.

Опция `--add-opens` использует тот же синтаксис, что и `--add-exports`, и открывает пакет для глубокой рефлексии, то есть все его типы и их члены доступны независимо от их модификаторов видимости. Из-за своего прямого отношения к рефлексии параметр более формально представлен в подразделе 12.2.2.

Тем не менее он используется для доступа к внутренним API, поэтому имеет смысл взглянуть на пример здесь. Довольно распространенный способ — инструменты, генерирующие экземпляры классов из других реализаций, скажем JAXB, создающий экземпляр `Customer` из XML-файла. Многие такие библиотеки полагаются на внутренние механизмы загрузки классов и для этого рефлексивно обращаются к непубличным членам класса JDK `ClassLoader`. Обратите внимание: в будущей

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

версии Java планируется удалить параметр `--illegal-access`, но Oracle еще не решил, в какой именно версии.

Если вы запустите такой код с параметром `--illegal-access=deny`, то это приведет к ошибке:

```
> Caused by: java.lang.reflect.InaccessibleObjectException:
>   Unable to make ClassLoader.defineClass accessible:
>   module java.base does not "opens java.lang" to unnamed module
```

Сообщение довольно ясно — решение состоит в том, чтобы использовать `--add-opens` при запуске приложения:

```
$ java
    --class-path ${jar-файлы}
    --illegal-access=deny
    --add-opens java.base/java.lang=ALL-UNNAMED
    ${основной_класс}
```

В отличие от `--illegal-access` и его текущего значения по умолчанию `permit`, параметры `--addexports` и `--add-opens` могут рассматриваться как «правильный способ» (или, скорее, «наименее сомнительный способ») для доступа к внутренним API. Разработчики намеренно формулируют эти параметры на основе требований своих проектов, и JDK поддерживает их в долгосрочной перспективе. Соответственно, система модулей не выдает предупреждений о разрешенном в итоге доступе.

Более того, они не позволяют `--illegal-access` выдавать предупреждения для пакетов, которые делают доступными. Если эти предупреждения вызывают в вашем коде ошибку, но вы не можете решить основную проблему, то экспорт и открытие пакетов таким образом устранят предупреждения. Если даже это не сработает (возможно, у вас нет доступа к командной строке), то посмотрите на решение на Stack Overflow: mng.bz/Bx6s. Только не говорите никому, откуда у вас эта ссылка.

ПРИМЕЧАНИЕ

Как я объяснил в подразделе 7.1.2, JDeps — отличный инструмент для поиска статического доступа к внутренним API JDK. Но как насчет рефлексивного доступа? Не существует надежного способа найти использование API, вызываемых через рефлексию, но иерархия вызовов на java.lang.reflect.AccessibleObject::setAccessible или полнотекстовый поиск для setAccessible обнаружит большинство из них в коде. Чтобы проверить проект в целом, запустите набор тестов или все приложение с параметром --illegal-access=debug или deny, чтобы обнаружить весь нелегальный доступ с помощью рефлексии.

7.1.5. Параметры компилятора и JVM для доступа к внутренним API

Проработав этот раздел, вы заслужили поощрение. Вся проблема внутренних API может показаться простой на первый взгляд, но, как только вы учтете наследие и проблемы совместимости экосистемы, она немного усложнится. Таблица 7.1 представляет обзор параметров и того, как они себя ведут.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Таблица 7.1. Сравнение различных механизмов, обеспечивающих доступ во время выполнения к внутренним API; разделение между статическим доступом (код, скомпилированный с такими классами или членами) и рефлексивным (с использованием API рефлексии)

Статический доступ				
Класс или член	**Публичный**		**Непубличный**	
Пакет	**Экспортированный**	**Неэкспортированный**	**Экспортированный**	**Неэкспортированный**
Строгая инкапсуляция	✓	×	×	×
По умолчанию в Java 9 из-за --illegal-access=permit	✓	✓	×	×
--illegal-access=warn	✓	✓	×	×
--illegal-access=debug	✓	✓	×	×
--illegal-access=deny	✓	×	×	×
--add-exports	✓	✓	×	×
--add-opens	✓	✓	×	×

Рефлексивный доступ				
Класс или член	**Публичный**		**Непубличный**	
Пакет	**Экспортированный**	**Неэкспортированный**	**Экспортированный**	**Неэкспортированный**
Строгая инкапсуляция	✓	×	×	×
По умолчанию в Java 9 из-за --illegal-access=permit	✓	×	До Java 9: в первый раз ⚠, иначе ×	×
--illegal-access=warn	✓	×	До Java 9: для всех ⚠, иначе ×	×
--illegal-access=debug	✓	×	До Java 9: для всех ⚠ с трассировкой стека, иначе ×	×
--illegal-access=deny	✓	×	×	×
--add-exports	✓	✓	×	×
--add-opens	✓	✓	✓	✓

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Помимо технических деталей, важно рассмотреть возможные стратегии, которые связывают приведенные выше и другие параметры на пути к совместимости с Java 9. Этим и занимается раздел 9.1. Если не предполагается указание параметров в командной строке (например, потому, что вы создаете исполняемый JAR-файл), то обратите особое внимание на подраздел 9.1.4 — он показывает три альтернативы данному подходу.

7.2. Улучшение разделенных пакетов

Проблемы с нелегальным доступом к внутренним API, с неразрешенными модулями JEE или с большинством других обсуждавшихся до сих пор изменений, какими бы раздражающими они ни были, имеют и некоторые плюсы: базовые концепции довольно просто понять; и благодаря точным сообщениям об ошибках проблемы легко распознать. Чего нельзя сказать о разделенных пакетах. В худшем случае единственный симптом, который вы увидите, — это ошибки компилятора или JVM, поскольку класс, явно находящийся в JAR на пути к классам, не может быть найден.

В качестве примера возьмем класс `MonitorServer`, который, помимо других аннотаций, использует `@Nonnull` из JSR 305. (Не беспокойтесь, если никогда его не видели, я объясню через минуту.) Вот что происходит при попытке скомпилировать это:

```
> error: cannot find symbol
>     symbol:   class javax.annotation.Nonnull
>     location: class monitor.MonitorServer
```

И это несмотря на то, что `jsr305-3.0.2.jar` находится в пути классов.

Что происходит? Почему некоторые типы не загружаются, хотя путь к классам содержит их? Критическое наблюдение таково: данные типы находятся в пакете, который содержится и в модуле. Теперь посмотрим, почему это имеет значение и приводит к тому, что классы не загружаются.

Когда разные артефакты содержат классы в одном и том же пакете (экспортированы или нет), говорят, что они *разделяют* пакет. Если хотя бы один из модульных JAR-файлов не экспортирует пакет, то это также называется *скрытым конфликтом пакетов*. Артефакты могут содержать классы с одинаковым полностью определенным именем, в этом случае разбиения перекрываются; или классы могут иметь разные имена и использовать только префикс имени пакета. Независимо от того, скрыты ли разделенные пакеты и перекрываются ли, эффекты, обсуждаемые в этом разделе, одинаковы. На рис. 7.2 показан разделенный и скрытый пакет.

> **Разделенные пакеты и модульное тестирование**
>
> Проблема разделенных пакетов — одна из двух причин, по которым модульные тесты, обычно размещаемые в другом дереве исходников, но в том же пакете, что и итоговый код, не создают собственный модуль. (Другая причина — строгая инкапсуляция, поскольку модульные тесты часто проверяют классы и методы, которые не являются публичными или отсутствуют в экспортированном пакете.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

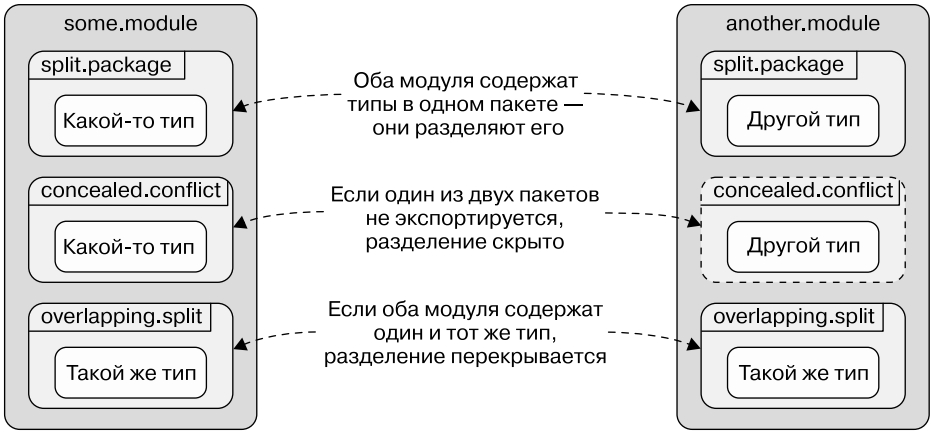


Рис. 7.2. Если два модуля содержат типы в одном пакете, то разделяют его

Обильные источники примеров разделенных пакетов — серверы приложений, которые обычно используют различные технологии JDK. Взять, скажем, сервер приложений JBoss и артефакт *jboss-jaxb-api_2.2_spec*. Он содержит классы, такие как `javax.xml.bind.Marshaller`, `javax.xml.bind.JAXB` и `javax.xml.bind.JAXBException`. Это явно перекрывает и, таким образом, разделяет пакет `javax.xml.bind`, содержащийся в модуле *java.xml.bind*. (Кстати, JBoss не делает ничего плохого: JAXB — автономная технология JEE, как описано в подразделе 6.1.1, и артефакт содержит ее полную реализацию.)

Пример неперекрывающегося и, как правило, более сомнительного разделенного пакета — JSR 305. Запрос спецификации Java (JSR) 305 хотел внести «аннотации для обнаружения дефектов программного обеспечения» в JDK. Он определился с несколькими аннотациями, такими как `@Nonnull` и `@Nullable`, которые должны были быть добавлены в пакет `javax.annotation`, создал эталонную реализацию, был успешно рассмотрен в соответствии с процессом сообщества Java (JCP), а затем — замолчал. Это был в 2006 году.

Сообществу, с другой стороны, понравились аннотации, поэтому инструменты статического анализа, такие как FindBugs, поддержали их и многие проекты приняли их. Хотя это не совсем стандартная практика, они обычно используются во всей экосистеме Java. Даже в Java 9 они не являются частью JDK, и, к сожалению, эталонная реализация помещает большинство аннотаций в пакет `javax.annotation`. Это создает неперекрывающееся разделение с помощью модуля *java.xml.ws.annotation*.

7.2.1. Что не так с разделенными пакетами

Какова же проблема с разделенными пакетами? Почему они могут привести к тому, что классы не будут найдены, хотя явно присутствуют? Ответ непрост.

Строго технический аспект разделенных пакетов заключается в следующем: весь механизм загрузки классов в Java был реализован при условии, что любое полностью

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

определенное имя класса уникально — по крайней мере в одном загрузчике классов, но поскольку по умолчанию существует всего один загрузчик классов для всего кода приложения, нет смысла ослаблять данное требование. Если загрузка классов Java не была переработана и реализована с нуля, то это запрещает перекрывающиеся разделения пакетов. (Раздел 13.3 показывает, как решить эту проблему путем создания нескольких загрузчиков классов.)

Другой технический аспект — команда JDK хотела использовать систему модулей для повышения производительности загрузки классов. Подраздел 6.2.1 описывает детали, но суть в том, что он полагается на знание, какому модулю принадлежит каждый пакет. Проще и эффективнее, если каждый пакет принадлежит только одному модулю.

Затем разделенные пакеты сталкиваются с важной целью системы модулей: надежной инкапсуляцией в пределах границ модуля. Что происходит, когда разные модули разделяют пакет? Разве они не могут получить доступ к пакетно-видимым классам и членам друг друга? Разрешение этого серьезно подорвет инкапсуляцию, но запрещение может столкнуться с вашим пониманием модификаторов видимости. Архитектурное решение, которое я предпочел бы не принимать.

Возможно, самый важный аспект является концептуальным. Предполагается, что пакет содержит согласованный набор классов с единственной целью. Так и модуль должен содержать согласованный набор пакетов с единственной, хотя и несколько большей, целью. В этом смысле два модуля, содержащие один и тот же пакет, имеют перекрывающиеся цели. Может быть, они должны быть одним модулем, тогда?..

Несмотря на отсутствие единого аргумента против разделенных пакетов, они имеют много свойств, которые нежелательны и будут способствовать несовместимости и двусмысленности. Следовательно, система модулей смотрит на них с подозрением и хочет предотвратить их использование.

7.2.2. Эффекты разделенных пакетов

Принимая во внимание несоответствия и неоднозначности, которые могут возникнуть при работе с разделенными пакетами, система модулей практически запрещает их:

- модуль не может читать один и тот же пакет из двух разных модулей;
- не допускается, чтобы два модуля в одном слое содержали один и тот же пакет (экспортированный или нет).

Что такое *слой*? Как объясняется в разделе 12.4, это контейнер, объединяющий загрузчик классов с целым модульным графом. До сих пор мы всегда неявно рассматривали однослойный случай, в котором второй пункт полностью включает в себя первый. Поэтому в случае добавления разных слоев запрещается разделение пакетов.

Как вы увидите далее, система модулей ведет себя по-разному в зависимости от того, где происходит разделение. После подробного изучения мы можем наконец перейти к исправлению разделения.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Разделения между модулями

Когда два модуля, такие как платформенный и модуль приложения, разделяют пакет, система модулей обнаружит данное действие и выдаст ошибку. Это может произойти во время компиляции или во время выполнения.

В качестве примера возьмем приложение *ServiceMonitor*. Как вы помните, модуль *monitor.statistics* содержит пакет `monitor.statistics`. Создадим пакет с тем же именем (и классом `SimpleStatistician`) в *monitor*. При компиляции этого модуля произойдет следующая ошибка:

```
> monitor/src/main/java/monitor/statistics/SimpleStatistician.java:1:
>     error: package exists in another module: monitor.statistics
>         package monitor.statistics;
>         ^
> 1 error
```

При попытке скомпилировать модуль с пакетом, который также экспортирован из необходимого модуля, компилятор замечает ошибку. Но что происходит, когда пакет не экспортируется, то есть появляется скрытый конфликт пакетов?

С целью выяснить это я добавил класс `monitor.Utils` в *monitor.statistics*. Это значит, я разделил пакет `monitor` между *monitor* и *monitor.statistics*. Разделение скрыто, поскольку *monitor.statistics* не экспортирует `monitor`.

В данной ситуации — и я считаю это немного удивительным — компиляция *monitor* работает. Сообщить об ошибке должна среда выполнения, что она покорно и делает сразу при запуске приложения:

```
> Error occurred during initialization of boot layer
> java.lang.reflect.LayerInstantiationException:
>     Package monitor in both module monitor.statistics and module monitor
```

То же самое верно, если два модуля (где ни один не запрашивает другой) содержат один и тот же пакет: не компилятор, а среда выполнения найдет ошибку.

Разделения между модулем и путем класса

Данная глава посвящена компиляции и запуску приложения с путем к классам в Java 9 или новее, поэтому вернемся к такому варианту использования. Интересно, что поведение системы модулей будет отличаться. Весь код из пути к классам попадает в *безымянный модуль* (подробнее об этом — в разделе 8.2); чтобы максимизировать совместимость, он, вообще говоря, не подвергается тщательному анализу и к нему не применяются проверки, связанные с модулями. Как следствие, система модулей не обнаружит разделенные пакеты и позволит скомпилировать и запустить приложение.

Поначалу это может звучать замечательно: меньше беспокойства. Увы, проблема все еще существует, она стала менее очевидной. И, возможно, даже ухудшилась.

Система модулей знает для каждого именованного модуля (в отличие от безымянного), какие пакеты он содержит и что каждый пакет принадлежит только одному модулю. Как я объяснил в подразделе 6.2.1, новая стратегия загрузки классов

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

использует эти знания; всякий раз при загрузке класса ищется модуль, содержащий пакет, и система пытается загрузить его оттуда. Если он содержит данный класс, то отлично; в противном случае результатом является `NoClassDefFoundError`.

Если пакет разделен между модулем и путем к классу, то загрузчик классов всегда *и только* заглядывает в модуль при загрузке классов из этого пакета (рис. 7.3).

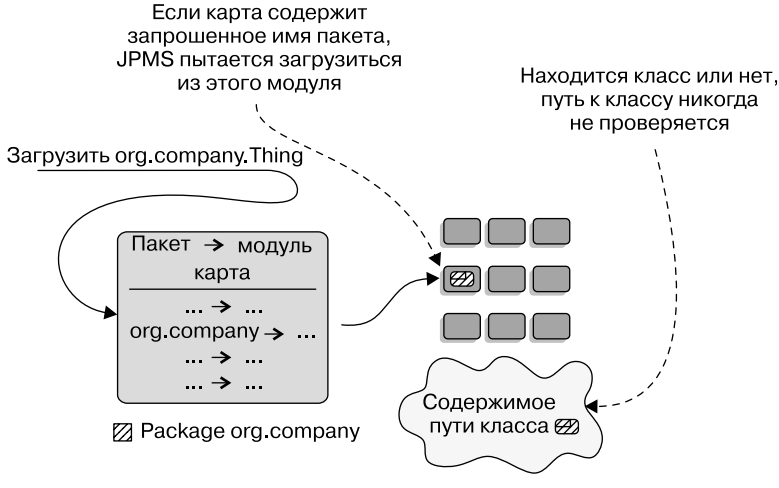


Рис. 7.3. Содержимое пути класса не подвергается проверкам модуля, а его пакеты не индексируются. Если он разделяет пакет с модулем, то загрузчик классов будет знать только о модуле и искать там классы. Здесь он ищет org.company и проверяет соответствующий модуль, игнорируя часть пакета, относящуюся к пути классов

Классы в пути класса фактически невидимы! Это верно для разделений между платформенными модулями и путем к классу и точно так же верно для модулей приложения (имеются в виду JAR-файлы, загруженные из пути модуля) и пути класса.

Да, вы поняли правильно. Если какой-то код содержит класс, скажем, из пакета `javax.annotation`, то загрузчик классов будет искать единственный модуль, который содержит данный пакет: *java.xml.ws.annotation*. Отсутствие там класса приведет к `NoClassDefFoundError`, даже если тот присутствует в пути к классам!

Как вы можете представить, произвольно пропущенные классы могут привести к некоторым головоломкам. Именно поэтому JEE-модули, которые способствуют разделению пакетов, не распределяются по умолчанию, как объясняется в разделе 6.1. Тем не менее эти модули могут создать самый странный случай разделения пакета.

Рассмотрим проект, который использует аннотации `@Generated` и `@Nonnull`. Первая присутствует в Java 8, а вторая взята из реализации JSR 305, которую проект имеет в своем пути к классам. Обе находятся в пакете `javax.annotation`. Что происходит при компиляции всего этого в Java 9 или более поздних версиях?

```
> error: cannot find symbol
>     symbol:   class Generated
>     location: package javax.annotation
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Так *класс Java* отсутствует? Да, поскольку это приходит из модуля JEE *java.xml.ws.annotation*, который по умолчанию не разрешен. Но сообщение об ошибке здесь другое: оно не намекает на решение. К счастью, вы обратили внимание ранее и знаете, что это можно исправить, добавив необходимый модуль с помощью `--add-modules java.xml.ws.annotation`. Тогда получится следующее:

```
> error: cannot find symbol
>     symbol:   class Nonnull
>     location: class MonitorServer
```

Компилятор нашел данный класс минуту назад — что не так? А вот теперь существует модуль, содержащий пакет `javax.annotation`, поэтому часть пути к классам становится невидимой.

Повторим (это также показано на рис. 7.4):

- первая ошибка была следствием тем, что модули JEE не были разрешены по умолчанию;
- вторая ошибка была вызвана тем, что система модулей проигнорировала часть пути класса разделенного пакета.

Это имеет смысл (верно?). Теперь, когда вы полностью понимаете, что происходит, перейдем к исправлению ситуации.

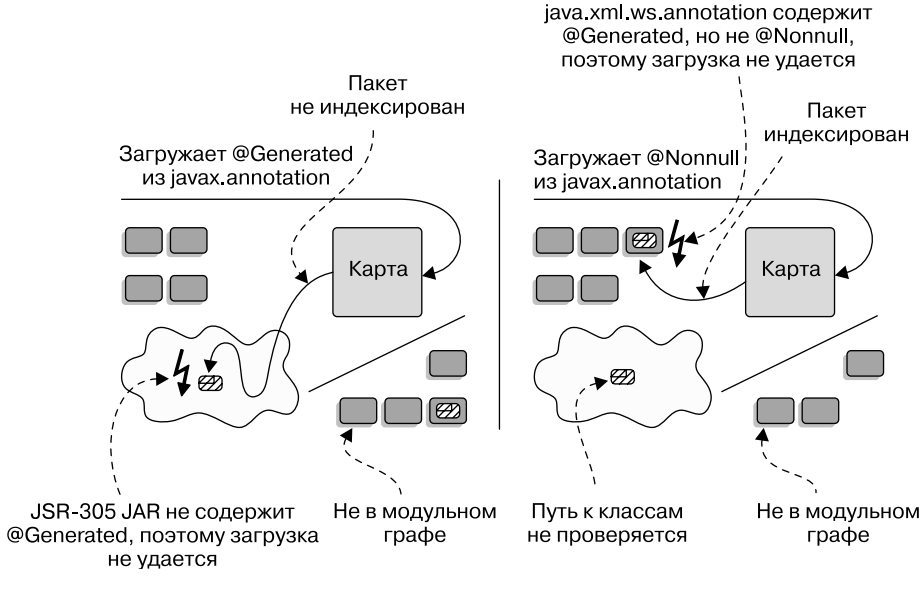


Рис. 7.4. При загрузке из одного пакета может произойти сбой по разным причинам. Слева JEE-модуль java.xml.ws.annotation не был добавлен, поэтому загрузка @Generated завершается неудачно, поскольку артефакт JSR 305 в пути к классам не содержит его. Справа был добавлен модуль, так что загрузка классов пытается загрузить оттуда все классы javax.annotation — даже @Nonnull, который содержит только JSR 305. В конце концов, оба подхода не позволяют загрузить все необходимые аннотации

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

7.2.3. Множество способов справиться с разделенными пакетами

Существует довольно много способов заставить разделенные пакеты работать. Вот они, рекомендую рассмотреть их в общем порядке.

- Переименуйте один из пакетов.
- Переместите все части разделенного пакета в один и тот же артефакт.
- Объедините артефакты.
- Оставьте оба артефакта в пути к классам.
- Обновите модуль JDK с помощью артефакта.
- Исправьте модуль с содержимым артефакта.

ПРИМЕЧАНИЕ

Только последние два способа применимы к типичному сценарию разделенных пакетов во время миграции, когда пакет разделяется между платформенным модулем и артефактом в пути к классам.

Первый подход работает, когда конфликт имен пакетов был случайным — он должен быть наиболее очевидным и использоваться по возможности. Когда разделение было сделано специально, это вряд ли сработает. В данном случае вы можете попытаться исправить разделение, переместив несколько классов или объединив артефакты. Первые три варианта — правильные, долгосрочные решения проблемы, но, очевидно, работают, только когда предполагается наличие контроля над артефактами разделения.

Если код разделения не принадлежит вам или решения неприменимы, то нужны другие варианты, которые заставят систему модулей работать, даже если пакет остается разделенным. Простое решение состоит в том, чтобы оставить оба артефакта в пути к классам, где они будут объединены в один и тот же безымянный модуль и станут вести себя так же, как и до Java 9. Это допустимая промежуточная стратегия на время, пока вы ожидаете хеширования конфликтующего проекта (-ов) и его исправления.

К сожалению, ни одно из решений, обсуждавшихся до сих пор, не применимо, когда часть разделения принадлежит модулю JDK, поскольку у вас нет прямого контроля над ним — чтобы преодолеть это разделение, нужны инструменты помощнее. Если повезет, то артефакт разделения будет состоять из нескольких классов, которые входят в случайный пакет JDK, и предоставится замена для всего обновляемого модуля JDK. В данном случае см. подраздел 6.1.3, в котором объясняется, как использовать `--upgrade-module-path`.

Если ничего из этого не помогло, то вы застряли на последнем и наиболее хакерском подходе: исправлении модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

7.2.4. Исправление модулей: последнее средство для обработки разделенных пакетов

Есть одна техника, которая может исправить почти каждый разделенный пакет, но к ней можно прибегать лишь в последнюю очередь: заставить систему модулей делать вид, что проблемные классы в пути к классам принадлежат модулю разделенного пакета. Параметр компилятора и среды выполнения `--patch-module ${`*`модуль`*`}=${`*`артефакт`*`}` объединяет все классы из `${`*`артефакт`*`}` в `${`*`модуль`*`}`. Есть несколько вещей, на которые стоит обратить внимание, но сначала рассмотрим пример, прежде чем доберемся до них.

Ранее мы рассмотрели пример проекта, который использует аннотации `@Generated` (из модуля *java.xml.ws.annotation*) и `@Nonnull` (из реализации JSR 305). Были обнаружены три момента:

- обе аннотации находятся в пакете `javax.annotation`, что создает разделение;
- нужно добавлять модуль вручную, поскольку это модуль JEE;
- это делает невидимой часть JSR 305 разделенного пакета.

Теперь вы знаете, что можно использовать `--patch-module` для исправления разделения:

```
javac
    --add-modules java.xml.ws.annotation
    --patch-module java.xml.ws.annotation=jsr305-3.0.2.jar
    --class-path 'libs/*'
    -d classes/monitor.rest
    ${исходные_файлы}
```

Таким образом, все классы в `jsr305-3.0.2.jar` становятся частью модуля *java.xml.ws.annotation* и могут быть загружены для успешной компиляции (или выполнения `java`). Ура!

Есть несколько аспектов, которые стоит иметь в виду. Для начала исправление модуля не добавляет его автоматически в модульный граф. Если это не требуется явно, то, возможно, его необходимо добавить с помощью `--add-modules` (см. подраздел 3.4.3).

Далее, классы, добавленные в модуль с `--patch-module`, подчиняются нормальным правилам доступности (см. раздел 3.3 и рис. 7.5):

- код, зависящий от таких классов, должен прочитать исправленный модуль, который должен экспортировать необходимые пакеты;
- кроме того, зависимости этих классов должны быть в экспортированных пакетах в модулях, считываемых исправленным модулем.

Это может потребовать манипулирования модульным графом с помощью параметров командной строки, таких как `--add-reads` (см. подраздел 3.4.4) и `--add-exports` (см. подраздел 11.3.4). Поскольку именованные модули не могут получить доступ к коду из пути к классам, может также потребоваться создать некоторые автоматические модули (см. раздел 8.3).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

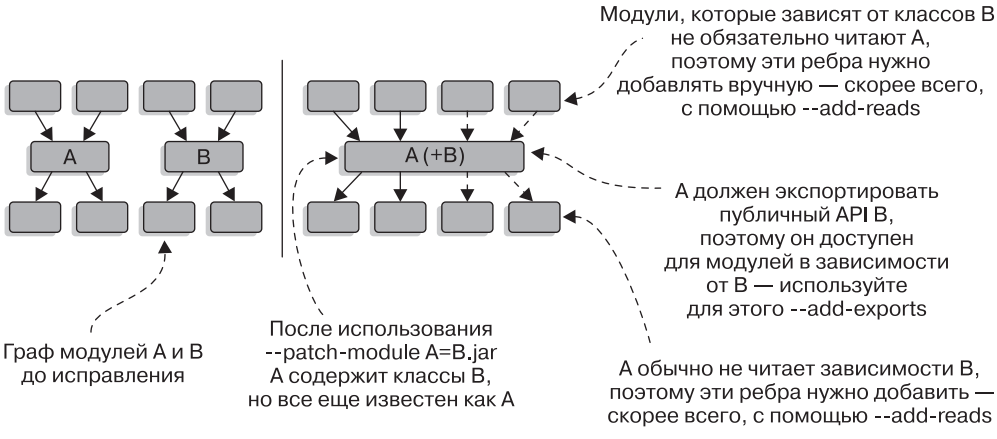


Рис. 7.5. Если классы модуля исправлены в другом модуле (здесь B в A), то входящие и исходящие зависимости исправленного модуля, а также экспорт пакетов должны быть отредактированы вручную, чтобы включенные классы работали должным образом

7.2.5. Поиск разделенных пакетов с помощью JDeps

Поиск разделенных пакетов методом проб и ошибок раздражает. К счастью, JDeps умеет сообщать о них. Приложение Г дает общее введение в инструмент; вам не нужно знать намного больше, поскольку разделенные пакеты включены практически в любой результат выполнения.

Посмотрим, что сообщает JDeps для приложения, использующего `javax.annotation.Generated` из *java.xml.ws.annotation* и `javax.annotation.Nonnull` из JSR 305. После копирования всех зависимостей в папку `lib` запускаем JDeps следующим образом:

```
$ jdeps -summary
    -recursive --class-path 'libs/*' project.jar

> split package: javax.annotation
>     [jrt:/java.xml.ws.annotation, libs/jsr305-3.0.2.jar]
>
# множество зависимостей проекта опущено
```

Это однозначно, верно? Если вам интересно, что именно зависит от разделенного пакета, то используйте `--package` и `-verbose:class`:

```
$ jdeps -verbose:class
    --package javax.annotation
    -recursive --class-path 'libs/*' project.jar

# разделенные пакеты опущены
# зависимости * из *javax.annotation опущены

> rest-1.0-SNAPSHOT.jar -> libs/jsr305-3.0.2.jar
>     monitor.rest.MonitorServer -> Nonnull jsr305-3.0.2.jar
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

7.2.6. Примечание о конфликте версий зависимостей

В подразделе 1.3.3 вы увидели: Java 8 не имеет встроенной поддержки для запуска нескольких версий одного и того же JAR-файла, например, если приложение транзитивно зависит от Guava 19 и 20. Всего несколькими страницами позже, в подразделе 1.5.6 вы узнали: к сожалению, система модулей этого не изменит. Из материала, который мы только что обсуждали о разделенных пакетах, должно быть понятно, почему это так.

Система модулей Java изменила стратегию загрузки классов (рассматривая конкретные модули вместо сканирования пути к классам), но не основные предположения и механизмы. Для каждого загрузчика классов все еще должен существовать только один класс с тем же полностью определенным именем, что делает невозможным несколько версий одного и того же артефакта. Более подробную информацию о поддержке версионности системой модулей см. в главе 13.

ПРИМЕЧАНИЕ

Вы узнали обо всех общих и некоторых необычных миграционных проблемах. Если хотите применить свои знания на практике и обновить проект до Java 9+, то перейдите к главе 9 — там обсуждается, как подойти к этому лучше всего. Как только ваше приложение заработает на Java 9+, станет возможным использование jlink для создания образов среды выполнения лишь с необходимыми модулями — см. раздел 14.1. Если вы заинтересованы в следующем шаге, который превращает существующую кодовую базу в модули, то переходите к главе 8.

Резюме

- Чтобы знать, как классы, от которых может зависеть ваш проект, могут быть доступны в системе модулей, важно понять, как они классифицируются в эпоху этой системы.
  - Все открытые классы в пакетах `java.*` или `javax.*` стандартизированы. Эти пакеты экспортируются модулями `java.*` и безопасны для использования, поэтому никаких изменений не требуется.
  - Открытые классы в некоторых пакетах `com.sun.*` поддерживаются Oracle. Такие пакеты экспортируются модулями `jdk.*` и в зависимости от них ограничивают базу кода конкретными поставщиками JDK.
  - Несколько отдельных классов в пакетах `sun.*` временно поддерживаются Oracle, пока в будущих версиях Java не будут введены замены. Они экспортируются с помощью *jdk-unsupported*.
  - Все остальные классы не поддерживаются и недоступны. Их можно применять с флагами командной строки, но код, который делает это, подвержен повреждению в JVM с различными второстепенными версиями или от разных поставщиков; таким образом, его использование нежелательно.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- Некоторые внутренние API были удалены, поэтому невозможно продолжать использовать их даже с параметрами командной строки.
- Несмотря на то что строгая инкапсуляция обычно запрещает доступ к внутренним API, для кода в пути к классам, который обращается к внутренним API JDK, делается исключение. Это значительно облегчит миграцию, но также усложнит поведение системы модулей.
  - Во время компиляции строгая инкапсуляция полностью активна и предотвращает доступ к внутренним API JDK. Тем не менее если некоторые API обязательны, то можно предоставить доступ к ним с помощью `--add-exports`.
  - Во время выполнения статический доступ к открытым классам в неэкспортированных пакетах JDK по умолчанию разрешен в Java 9–11. Это повышает вероятность того, что существующие приложения будут работать естественным образом, но ситуация изменится в будущих выпусках.
  - Рефлексивный доступ ко всем внутренним API JDK разрешен по умолчанию, но приведет к предупреждению либо при первом доступе к пакету (по умолчанию), либо при каждом доступе (с `--illegal-access=warn`). Лучший способ проанализировать это — `--illegal-access=debug`, который включает трассировку стека в каждом предупреждении.
  - Более строгое поведение для статического и рефлексивного доступа возможно с `--illegal-access=deny`, при использовании `--add-exports` и `--add-opens` в случае надобности, для доступа к критически необходимым пакетам. Чем раньше вы это сделаете, тем легче будет перейти на будущие обновления Java.
- Система модулей запрещает двум модулям (в одном слое) содержать один и тот же пакет — экспортированный или нет. Однако наличие кода в пути к классам не проверяется, поэтому возможно разделение нераскрытого пакета между платформенным модулем и кодом в пути к классам.
- Если пакет разделен между модулем и путем класса, то часть пути класса практически невидима, что приводит к неожиданным ошибкам во время компиляции и выполнения. Лучшее решение — удалить разделение, но при отсутствии возможности сделать это платформенный модуль, о котором идет речь, можно либо заменить артефактом разделения на `--upgrade-module-path` (если это обновляемый модуль), либо исправить его содержимое с помощью `--patch-module`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

8 Постепенная модуляризация существующих проектов

В этой главе

- Работа с безымянными модулями.
- Помощь в модуляризации с помощью автоматических модулей.
- Наращиваемая модульная кодовая база.
- Смешивание пути класса и пути модуля.

В зависимости от того, насколько гладко прошел ваш переход на Java 9+ (см. главы 6 и 7), вы можете столкнуться с некоторыми из более неприятных последствий внедрения системы модулей в достаточно развитую экосистему. Хорошая новость — это того стоило! Как я кратко показал в подразделе 1.7.1, Java 9+ может многое предложить, помимо системы модулей. Если вы в состоянии повысить требования Java к своему проекту до 9, то можете сразу начать использовать их.

Можно наконец начать модуляризацию вашего проекта. Превратив артефакты в модульные JAR-файлы, вы и ваши пользователи можете получить выгоду от надежной конфигурации (см. подраздел 3.2.1), надежной инкапсуляции (см. подраздел 3.3.1), развязки с помощью сервисов (см. главу 10), образов среды выполнения, включая целые приложения (см. раздел 14.2), и других связанных с модулем ценных свойств. Как будет показано в подразделе 9.3.4, можно модулировать даже проекты, которые работают на Java 8 и более ранних версиях.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Существует два способа сделать JAR модульными:

- подождать, пока все зависимости не станут модульными, а затем создать дескрипторы модулей для всех артефактов сразу;
- начать с ранней модуляризации только ваших артефактов, возможно, всего парочки за раз.

С учетом того, что обсуждается в главах 3, 4 и 5, реализация первого варианта должна быть простой. Вам могут понадобиться некоторые более расширенные функции системы модулей, которые представлены в главах 10 и 11, но, кроме этого, можно пойти дальше: создать декларацию модуля для каждого нового артефакта и смоделировать их отношения, как было изучено ранее.

Возможно, ваш проект находится глубоко в дереве зависимостей и не придется ждать, пока все зависимости станут модульными. Или, вероятно, ваш проект слишком велик, чтобы превратить все артефакты в модули за один раз. В этих случаях вас может заинтересовать второй вариант, который позволяет постепенно наращивать модульные артефакты независимо от того, являются ли их зависимости модульными или простыми JAR-файлами.

Возможность использовать модульные и немодульные артефакты бок о бок не только важна для отдельных проектов, но и означает, что экосистема в целом может добавлять модули независимо друг от друга. Без этого модуляризация экосистемы могла бы занять несколько десятков лет — а так любой сможет сделать это в течение одного десятилетия.

Эта глава посвящена функциям, которые позволяют постепенно наращивать модульность существующих проектов: мы начнем с обсуждения комбинации путей класса и модуля, затем исследуем безымянный модуль и в заключение рассмотрим автоматические модули. По завершении ваш проект или его части получат выгоду от применения системы модулей, несмотря на потенциально немодулированные зависимости. Вы также будете хорошо подготовлены к главе 9, в которой рассматриваются стратегии модуляризации приложений.

8.1. Почему наращивание модульности необязательно

Прежде чем перейти к тому, как постепенно наращивать модульность проекта, я хочу разобраться, почему это необязательно. Системы модулей обычно требуют, чтобы все было модульно. Но если они не появляются вовремя (например, JPMS) или используются лишь небольшой частью своей экосистемы (скажем, модулями OSGi или JBoss), то вряд ли ожидается, что это произойдет. Нужно найти способ взаимодействовать с немодульными артефактами.

В данном разделе мы сначала подумаем о развитии событий, если каждый JAR должен быть модульным для запуска на Java 9+; это приводит к выводу, что должна быть возможность смешивать простые JAR и модули (см. подраздел 8.1.2). Затем я покажу, как использование пути класса и пути модуля бок о бок позволяет применить данный подход «смешай и соедини» (см. подраздел 8.1.3).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

8.1.1. Если бы требовалось, чтобы каждый JAR был модульным

Если бы JPMS была строгой и требовала, чтобы все было модульно, то ею можно было бы пользоваться, только если все JAR содержали бы дескриптор модуля. А поскольку система модулей — неотъемлемая часть Java 9+, в результате ее нельзя было бы обновить, не модуляризовав весь код и зависимости. Представьте последствия, будь так на самом деле.

Одни проекты могут обновиться до Java 9+ на ранней стадии, что заставит всех их пользователей модуляризовать свои кодовые базы или прекратить взаимодействие с проектом. Другие могут не захотеть форсировать данное решение или же могут иметь другие причины не перепрыгивать на модули, а это сдерживает их пользователей. Я не хотел бы, чтобы у моего проекта были зависимости, которые принимали противоположные решения. Что я мог бы сделать?

С другой стороны, некоторые проекты будут поставлять отдельные варианты с дескрипторами модулей и без них, вследствие чего им придется задействовать два совершенно непересекающихся набора зависимостей (один с дескрипторами модулей и один без них). Кроме того, если бы они не вносили исправления в старые и новые версии, то пользователи были бы вынуждены одновременно выполнять много (вероятно, требующих много времени) обновлений, чтобы иметь возможность перейти на Java 9+. И это даже без учета проектов, которые больше не поддерживаются и быстро стали бы непригодными для Java 9+, даже если бы сами не имели никаких зависимостей.

Единственный способ избежать бесполезных усилий и глубокого раскола заключается в том, чтобы в сообществе наступил день, когда *все проекты* обновились до Java 9+ и начали выпускать модульные JAR-файлы. Но не существует способа сделать это. И как бы мы это ни скрывали, любой человек, запускающий JAR, должен знать, для какой версии Java тот был создан, поскольку не будет работать на 8 *и* 9. Итак, у нас большие проблемы!

8.1.2. Смешивание и соединение простых JAR-файлов с модулями

Чтобы обойти данную проблему, система модулей должна предлагать способ запуска немодульного кода поверх модульной JVM. Во введении к главе 6 я объясняю, что это действительно так и простые JAR-файлы в пути к классам работают так же, как и до Java 9+. (Как говорится в главах 6 и 7, код, который они содержат, может не работать, но это другой вопрос.) В разделе 8.2 освещается, как работает *режим пути к классам*.

Тот факт, *что* он работает, уже является важным открытием: система модулей может обрабатывать немодульные артефакты и знает, как перемещаться по границе между ними и явными модулями. Это хорошая и не единственная новость: данная граница не является незыблемой. Не нужно отделять JAR-файлы приложения от модулей JVM. Как показано на рис. 8.1 и исследуется на протяжении этой главы, система модулей позволяет перемещать эту границу, а также смешивать и сопоставлять модульные и немодульные JAR-файлы приложения с платформенными модулями, как того требуют ваши проекты.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

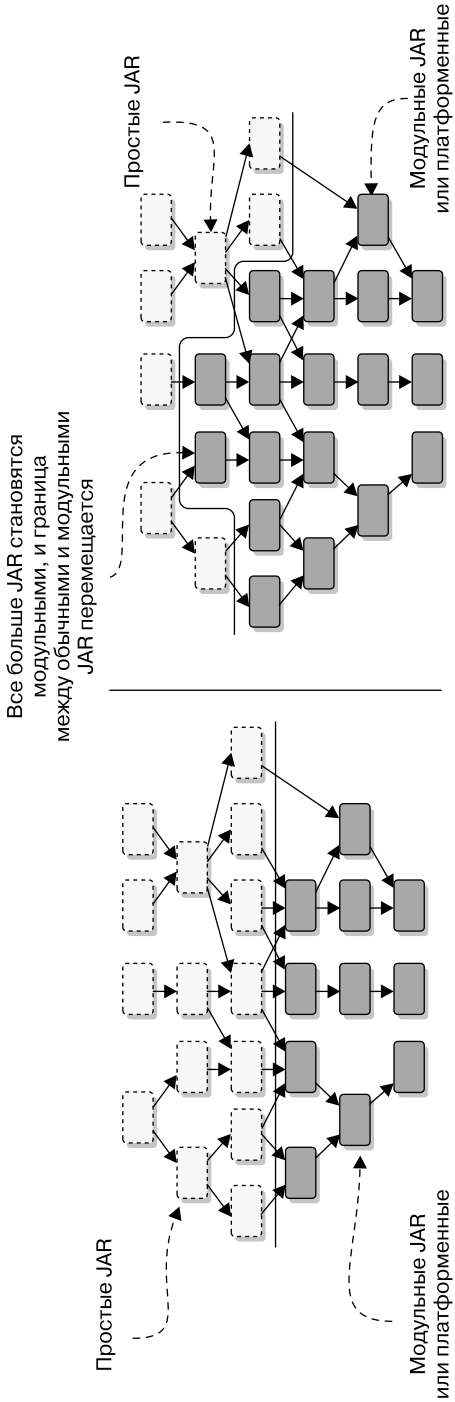


Рис. 8.1. Система модулей позволяет выполнять немодульный код на модульном JDK *(слева)*. Что еще более важно, она предоставляет инструменты для перемещения этой границы *(справа)*

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

8.1.3. Технические основы наращиваемой модульности

Основной принцип, который делает вероятной постепенную модуляризацию, заключается в том, что пути классов и модулей могут использоваться бок о бок. Нет необходимости перемещать все JAR приложения из пути класса в путь модуля за один раз. Вместо этого в существующих проектах рекомендуется начинать с пути класса, а затем медленно перемещать свои артефакты в путь модуля по мере приложения усилий по модуляризации.

Использование обоих путей одновременно с простыми и модульными JAR-файлами требует четкого понимания того, как соотносятся эти концепции. Можно подумать, будто JAR без дескриптор модуля находятся в пути класса, а модульные — в пути модуля. Хотя я никогда так не говорил, вас простят за то, что вы читали между строк. Тем не менее данная теория ошибочна, и сейчас самое время от нее отказаться.

Два механизма опровергают эту теорию и делают возможным наращивание модуляризации:

- *безымянный модуль* неявно создается системой модулей со всем содержимым, загруженным из пути классов. В нем живет хаос пути классов (в разделе 8.2 это объясняется подробно);
- система модулей создает *автоматический модуль* для каждого простого JAR-файла, найденного в пути модуля (этой концепции посвящен раздел 8.3).

Путь к классу не делает различий между обычными и модульными JAR-файлами: если те находятся в пути к классам, то попадают в безымянный модуль. Точно так же путь модуля еле различает простые и модульные JAR-файлы: если те находятся в пути к модулю, то рассматриваются как именованные модули. (Для простых JAR система модулей создает автоматический модуль; для модульных — явный модуль в соответствии с описанием.)

Чтобы понять остальную часть этой главы, а также провести модуляризацию, важно полностью усвоить данное поведение. Таблица 8.1 показывает двумерную переработку. Не тип JAR (простой или модульный), а путь, в котором он размещен (путь класса или путь модуля), определяет, станет ли он частью безымянного или именованного модуля.

Таблица 8.1. Не тип JAR, а путь, в котором он размещен, определяет, где будет класс: в именованном модуле или в безымянном

JAR	Путь класса	Путь модуля
Обычный	Безымянный модуль (см. раздел 8.2)	Автоматический модуль (см. раздел 8.3)
Модульный	—	Явный модуль (см. подраздел 3.1.4)

При принятии решения о том, помещать ли JAR в путь класса в или путь модуля, дело не в том, откуда берется код (является ли JAR модульным), а в том, где он необходим (в безымянном или именованном модуле). Путь класса предназначен для

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

кода, который вы хотите добавить в комок грязи, а путь модуля — для кода, предназначенного стать модулем.

Но как решить, куда отправить код? Обычно безымянный модуль касается *совместимости*, позволяя проектам, использующим путь классов, работать на Java 9+; тогда как автоматические модули — это *модульность*, позволяющая проектам использовать систему модулей, даже если зависимости еще не модульны.

Для более подробного ответа стоит присмотреться к безымянным и автоматическим модулям. Затем, в главе 9, мы определим более крупные стратегии модуляризации. Если вас интересует, стоит ли модулировать существующий проект, то обратитесь к подразделу 15.2.1.

ПРИМЕЧАНИЕ

Ваш инструмент сборки может принять многие из этих решений за вас. Тем не менее все равно есть вероятность оказаться в ситуациях, когда что-то пошло не так, и в данном случае вы можете применить материал этой главы для правильной настройки сборки.

8.2. Безымянный модуль, он же путь к классу

Есть один аспект, который я еще не объяснил подробно: как система модулей и путь к классам работают вместе? Первая часть книги дает четкое представление о том, как модульные приложения размещают все в пути модуля и работают с модульным JDK. Затем последовали главы 6 и 7, посвященные компиляции немодульного кода и запуску приложений из пути классов. Но как содержимое пути к классу взаимодействует с системой модулей? Какие из них разрешены и как? Почему содержимое пути класса может получить доступ ко всем платформенным модулям? Безымянный модуль отвечает на эти вопросы.

Их изучение имеет не только академическую ценность. Если приложение не является достаточно маленьким, то, вероятно, не может быть полностью модульным; но наращиваемая модульность включает в себя смешивание JAR-файлов и модулей, путей классов и модулей. Это очень важно для понимания основных деталей того, как работает режим пути к классам системы модулей.

ПРИМЕЧАНИЕ

Механизмы, окружающие безымянный модуль, обычно применяются во время компиляции и выполнения, но всегда упоминание и того и другого излишне раздувает текст. Вместо этого я описываю поведение во время выполнения и упоминаю время компиляции только тогда, когда поведение отличается.

Безымянный модуль содержит все одномодульные классы, которые:

- во время компиляции компилируются, если не содержат дескриптор модуля;
- во время компиляции и выполнения загружаются из пути к классам.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Как описано в подразделе 3.1.3, все модули имеют три основных свойства, и это также верно для безымянного модуля:

- *имя* — у безымянного модуля его нет (что имеет смысл, верно?), то есть никакой другой модуль не может упомянуть его в своих объявлениях (например, запросить его);
- *зависимости* — безымянный модуль считывает все другие модули, из которых состоит граф;
- *экспорты* — безымянный модуль экспортирует все свои пакеты, а также открывает их для рефлексии (подробности об открытых пакетах и модулях см. в разделе 12.2).

В отличие от безымянного модуля все остальные называются *именованными*. Сервисы, предоставляемые в `META-INF/services`, доступны для `ServiceLoader`. В главе 10 представлено введение в сервисы, и, в частности, в подразделе 10.3.6 описано их взаимодействие с безымянным модулем.

Хотя она и не настолько проста, концепция безымянного модуля имеет смысл. Здесь есть упорядоченный граф модулей, а там, немного в стороне, есть хаос путей к классам, сосредоточенный в собственном модуле, доступном для всех, с некоторыми особыми свойствами (рис. 8.2). (Чтобы не усложнять вопросы сверх необходимого, я не говорил ранее, но безымянный модуль лежит в основе глав 6 и 7, где можно заменить любое вхождение содержимого *пути класса безымянным модулем*.)

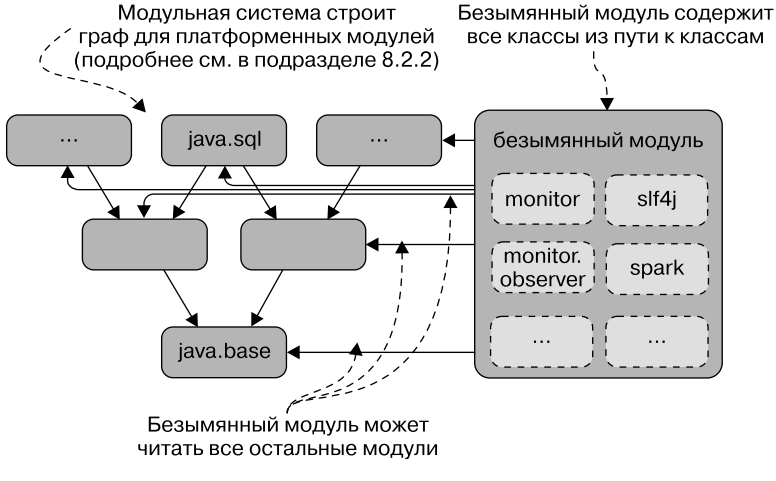


Рис. 8.2. Запущенная со всеми JAR-файлами приложения в пути к классам, система модулей строит граф из платформенных модулей *(слева)* и распределяет все классы из пути к классам к безымянному модулю *(справа)*, который может читать все другие модули

Вернемся к приложению *ServiceMonitor* и предположим, что оно было написано до Java 9. Код и его организация идентичны тем, которые мы обсуждали в предыдущих главах, но в нем отсутствуют объявления модулей, поэтому создаются простые JAR-файлы вместо модульных.

Предполагая, что папка `jars` содержит все JAR-файлы приложения, а `libs` — все зависимости, можно запустить приложение следующим образом:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
$ java --class-path 'jars/*':'libs/*' monitor.Main
```

Это работает в Java 9+ и, кроме альтернативной формы параметра `--class-path`, делает то же самое, что в Java 8 и более ранних версиях. На рис. 8.2 показан граф модулей, который система модулей создает для данной конфигурации запуска.

Теперь, понимая это, вы хорошо подготовлены к запуску простых, одномодульных приложений из пути класса. Помимо этого базового варианта использования и особенно при медленной модуляризации приложения тонкости безымянного модуля становятся актуальными, вследствие чего мы рассмотрим их далее.

8.2.1. Хаос пути к классам, захваченного безымянным модулем

Основная цель безымянного модуля — захватить содержимое пути класса и заставить его работать в системе модулей. Поскольку в пути к классам никогда не было границ между JAR, нет смысла устанавливать их сейчас; поэтому наличие единственного безымянного модуля для всего пути класса — разумное решение. Внутри него, как и в пути к классам, доступны все публичные классы и не существует концепции разделенных пакетов.

Отличительная роль безымянного модуля и его ориентация на обратную совместимость придают ему несколько особых свойств. В разделе 7.1 вы заметили, что во время выполнения строгая инкапсуляция платформенных модулей в основном отключена для кода в безымянном модуле (по крайней мере в Java 9, 10 и 11). Когда мы обсуждали разделенные пакеты в разделе 7.2, обнаружилось, что безымянный модуль не сканируется, поэтому разделения пакетов между ним и другими модулями не обнаруживаются, а часть пути к классам недоступна.

Одна деталь, немного нелогичная, из-за которой легко ошибиться, — это то, что составляет безымянный модуль. Кажется очевидным, что модульные JAR-файлы становятся модулями и, следовательно, простые JAR-файлы переходят в безымянный модуль, верно? Как объяснено в подразделе 8.1.3, это неправильно: безымянный модуль отвечает за *все JAR в пути к классам*, модульные или нет.

Как следствие, модульные JAR-файлы не обязательно должны загружаться как модули! Если библиотека начинает предоставлять модульные JAR-файлы, то ее пользователи ни в коем случае не должны применять их в качестве модулей. Вместо этого они могут оставить их в пути к классам, где их код объединен в безымянный модуль. Как более подробно объясняется в разделе 9.2, это позволяет экосистеме модулировать части приложений практически независимо друг от друга.

В качестве примера запустим полностью модульную версию *ServiceMonitor*, один раз из пути класса и один раз из пути модуля:

```
$ java --class-path 'mods/*':'libs/*' -jar monitor
$ java --module-path mods:libs --module monitor
```

И тот и другой работают нормально и без каких-либо явных различий.

Один из способов увидеть, как система модулей обрабатывает оба случая, — использовать API, который мы более подробно рассмотрим в подразделе 12.3.3. Можно

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

вызвать `getModule` для класса, чтобы получить модуль, которому он принадлежит, а затем `getName` для этого модуля, чтобы получить его имя. Для безымянного модуля `getName` возвращает `null`.

Добавим следующие строки кода в `Main`:

```
String moduleName = Main.class.getModule().getName();
System.out.println("Module name: " + moduleName);
```

При запуске из пути класса в результате получаем `Module name: null`, что указывает на то, что класс `Main` оказался в безымянном модуле. При запуске из пути модуля получим ожидаемое `Module name: monitor`.

В подразделе 5.2.3 обсуждается, как система модулей инкапсулирует ресурсы в пакетах. Это лишь частично относится к безымянному модулю: в модуле нет ограничений доступа (вследствие чего все JAR-файлы из пути к классам могут обращаться к ресурсам друг друга), а безымянный модуль открывает все пакеты для рефлексии (так что все модули могут получать доступ к ресурсам из JAR-файлов в пути к классам). Тем не менее строгая инкапсуляция применяется для доступа из безымянного к именованному модулю.

8.2.2. Разрешение модулей для безымянного модуля

Важный аспект отношения безымянного модуля к остальной части графа — то, какие другие модули он может читать. Выше описано, что он может читать все модули, которые составляют граф. Но какие именно?

Напомню материал подраздела 3.4.1: разрешение модуля строит граф, начиная с корневых модулей (в частности, исходного), а затем итеративно добавляя все их прямые и переходные зависимости. Как это будет работать, если код в процессе компиляции или метод `main` приложения находится в безымянном модуле, как в случае запуска приложения из пути к классам? В конце концов, простые JAR-файлы не выражают никаких зависимостей.

Если начальный модуль является безымянным, то все системные модули (в первую очередь все модули *java.** и *jdk.**) и все модули на пути к модулю обновления, которые экспортируют хотя бы один пакет без квалификации, становятся корневыми.

- Точный набор модулей *java.**, которые становятся корневыми, зависит от наличия модуля *java.se* (модуля, представляющего весь API Java SE, — он присутствует в полных образах Java, но может отсутствовать в пользовательских образах среды выполнения, созданных с помощью `jlink`):
  - если *java.se* является обозреваемым, то становится корневым;
  - если это не так, то каждый системный модуль *java.** и модуль *java.** из обновленного пути к модулю, который экспортирует хотя бы один пакет без квалификации (то есть без ограничений на то, кто может получить доступ к пакету, см. раздел 11.3), становится корневым.
- Помимо модулей *java.**, каждый другой системный модуль и модуль из обновленного пути модуля, который не является инкубационным модулем и экспортирует

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

хотя бы один пакет без квалификации, становится корневым. Особенно это относится к модулям *jdk.** и *javafx.**.

- Модули, определенные с помощью `--add-modules` (см. подраздел 3.4.3), всегда являются корневыми.

Это кажется сложным (см. рис. 8.3 для визуализации), но может стать важным в крайних случаях. Эмпирическое правило о том, что все системные модули, кроме JEE и инкубационных, разрешены, должно охватывать как минимум 90 % случаев.

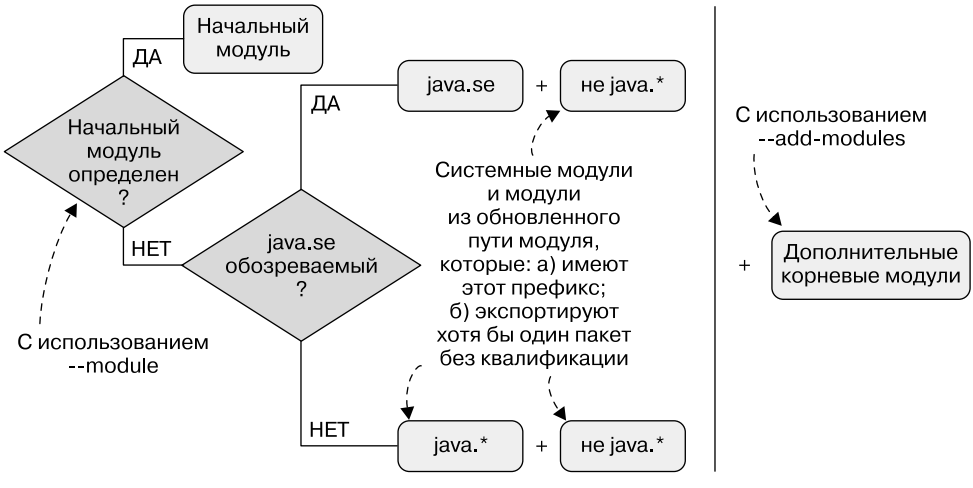


Рис. 8.3. Какие модули становятся корневыми для разрешения модулей (см. подраздел 3.4.1), зависит от того, был ли начальный модуль определен с помощью --module (если нет, то безымянный модуль становится исходным), и от того, является ли java.se обозреваемым. В любом случае модули, определенные с помощью --add-modules, всегда корневые

В качестве примера можно запустить `java --show-module-resolution` и наблюдать первые несколько строк вывода:

```
> root java.se jrt:/java.se
> root jdk.xml.dom jrt:/jdk.xml.dom
> root javafx.web jrt:/javafx.web
> root jdk.httpserver jrt:/jdk.httpserver
> root javafx.base jrt:/javafx.base
> root jdk.net jrt:/jdk.net
> root javafx.controls jrt:/javafx.controls
> root jdk.compiler jrt:/jdk.compiler
> root oracle.desktop jrt:/oracle.desktop
> root jdk.unsupported jrt:/jdk.unsupported
```

Это не весь вывод, и порядок в вашей системе может быть другим. Но, начав сверху, можно заметить, что *java.se* — единственный модуль *java.**. Кроме того, есть несколько модулей *jdk.** и *javafx.** (определяют *jdk.unsupported* из подраздела 7.1.1), а также модуль *oracle.** (понятия не имею, что он делает).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

Обратите внимание: с безымянным модулем в качестве исходного набор корневых модулей всегда является подмножеством системных модулей, содержащихся в образе выполнения. Модули, присутствующие в пути модуля, никогда не будут разрешены, если явно не добавлены с помощью --add-modules. В случае установления пути к модулю вручную, чтобы он содержал именно те модули, которые необходимы, можно добавить их все с помощью --add-modules ALL-MODULE-PATH, как описано в подразделе 3.4.3.

Можно легко наблюдать это поведение, запустив *ServiceMonitor* из пути к модулю без определения исходного модуля:

```
$ java --module-path mods:libs monitor.Main

> Error: Could not find or load main class monitor.Main
> Caused by: java.lang.ClassNotFoundException: monitor.Main
```

Выполнение той же команды с параметром `--show-module-resolution` подтверждает, что модули *monitor.** не разрешены. Исправить это можно с помощью `--add-modules monitor`, тогда модуль *monitor* добавляется в список корневых, или `--module monitor/monitor.Main`, в этом случае *monitor* становится единственным корневым модулем (начальным).

8.2.3. Зависимость от безымянного модуля

Одна из основных целей системы модулей — надежная конфигурация: модуль должен выражать свои зависимости, а система должна быть в состоянии гарантировать их присутствие. Мы установили это в разделе 3.2 для явных модулей с дескриптором. Что произойдет, если вы попытаетесь расширить надежную конфигурацию до пути к классам?

Проведем мысленный эксперимент. Представьте, что модули могут зависеть от содержимого пути к классам, скажем, с чем-то вроде `requires class-path` в дескрипторе. Какие гарантии может дать система модулей для такой зависимости? Выяснилось, что почти никаких. Пока существует хотя бы один класс из пути к классам, система модулей будет предполагать, что зависимость присутствует. Это не поможет (рис. 8.4).

Хуже того, это серьезно подорвет надежную конфигурацию, поскольку вы можете оказаться в зависимости от модуля с `requires class-path`. Хорошо, он не содержит никакой информации — что *именно* нужно положить в путь классу (опять же см. рис. 8.4)?

Развивая эту гипотезу, представьте, что два модуля, *com.framework* и *org.library*, зависят от одного и того же третьего модуля, скажем SLF4J. Один из них объявил зависимость до того, как SLF4J был модульным, и, следовательно, имеет `requires class-path`; другой объявил о своей зависимости от модульного SLF4J и, следовательно, имеет `requires org.slf4j` (при условии, что это имя модуля). Теперь, в какой путь кто-либо в зависимости от *com.framework* и *org.library* поместит JAR

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

SLF4J? Что бы они ни выбрали, система модулей должна была определить, что одна из двух переходных зависимостей не была заполнена. На рис. 8.5 показана эта гипотетическая ситуация.

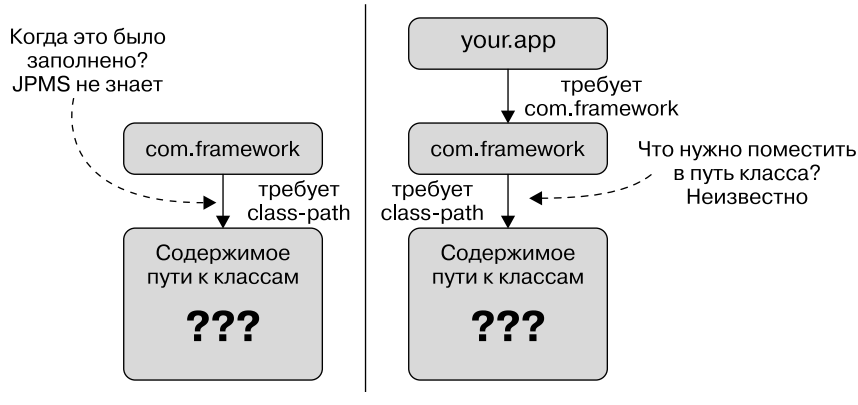


Рис. 8.4. Если com.framework зависит от некоего содержимого пути к классу с гипотетическим requires class-path, то система модулей не может определить, было ли выполнено это требование *(слева)*. Создав свое приложение на данной платформе, вы не будете знать, как заполнить эту зависимость *(справа)*

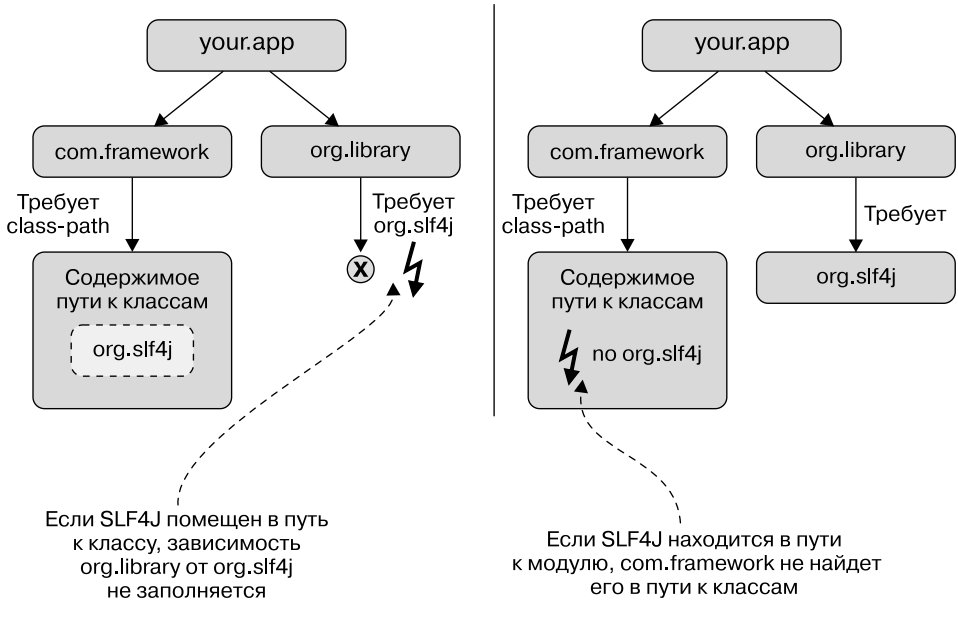


Рис. 8.5. Если com.framework зависит от SLF4J с гипотетическим requires class-path и org.library запрашивает его как модуль с requires org.slf4j, то не существовало бы способа удовлетворить оба требования. Независимо от того, был ли SLF4J размещен в пути к классам *(слева)* или в пути к модулям *(справа)*, одна из двух зависимостей будет считаться незаполненной

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Размышление над всем вышесказанным приводит к выводу, что зависимость от произвольного содержимого пути к классу — не очень хорошая идея, если нужны надежные модули. И по этой причине `requires class-path` не существует.

Как лучше всего выразить, что модуль, в котором хранится содержимое пути к классу, не может зависеть от чего-либо? В системе модулей, использующей имена для ссылки на другие модули? Невозможность дать этому модулю имя, сделав его *безымянным*, звучит оправданно.

И вот что мы имеем: у безымянного модуля нет имени, поскольку ни один модуль не должен ссылаться на него в директиве `requires` — или любой другой директиве, если уж на то пошло. Без `requires` не будет ребра читабельности, а без него код в безымянном модуле недоступен для других модулей.

Таким образом, чтобы явный модуль зависел от артефакта, этот артефакт должен находиться в пути модуля. Как уже упоминалось в подразделе 8.1.3, это может означать, что при размещении простых JAR-файлов в пути модулей они превращаются в автоматические модули — концепция, которую мы рассмотрим далее.

8.3. Автоматические модули: простые JAR в пути модуля

Долгосрочная цель любых усилий по модуляризации — обновить простые JAR-файлы до модульных и переместить их из пути классов в путь модуля. Один из способов добиться этого — подождать, пока все зависимости станут модульными, а затем модульно преобразовать проект — так выглядит нисходящий подход. Однако он может занять много времени, поэтому система модулей также допускает модульность по направлению сверху вниз.

Раздел 9.2 объясняет оба подхода в деталях, но для того, чтобы нисходящий подход работал, вам нужен новый ингредиент. Подумайте об этом: как можно объявить модуль, если зависимости представлены в виде простых JAR-файлов? Как вы заметили в подразделе 8.2.3, при размещении их в пути к классам они окажутся в безымянном модуле и другой модуль не сможет получить к ним доступ. Но вы обратили внимание на подраздел 8.1.3, поэтому уже знаете, что простые JAR-файлы также могут находиться в пути модуля, где система модулей автоматически создает модули для них.

ПРИМЕЧАНИЕ

> Механизмы, окружающие автоматические модули, обычно применяются во время компиляции и выполнения. Как я уже говорил ранее, постоянное упоминание и того и другого дает мало информации и затрудняет чтение текста.

Для каждого JAR в пути модуля, у которого нет дескриптора, система модулей создает *автоматический модуль*. Как и любой другой именованный, он имеет три основных свойства (см. подраздел 3.1.3):

- *имя* — имя автоматического модуля может быть определено в манифесте JAR с заголовком `Automatic-Module-Name`. Если он отсутствует, то система модулей генерирует имя из имени файла;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- *зависимости* — автоматический модуль считывает все остальные модули, которые входят в граф, включая безымянный (как вы скоро увидите, это важно);
- *экспорты* — автоматический модуль экспортирует все свои пакеты, а также открывает их для рефлексии (подробности об открытых пакетах и модулях см. в разделе 12.2).

Кроме того, исполняемые JAR-файлы приводят к появлению *исполняемых модулей*, чей основной класс помечен как `.described`, что описано в подразделе 4.5.3. Сервисы, предоставляемые в `META-INF/services`, доступны для `ServiceLoader` — в главе 10 представлено введение в сервисы, и, в частности, в разделе 10.3.6 описано их взаимодействие с автоматическими модулями.

Еще раз предположив, что *ServiceMonitor* еще не был модульным, вы тем не менее можете поместить его артефакты в путь модуля. Если каталог `jars-mp` содержит `monitor.jar`, `monitor.observer.jar` и `monitor.statistics.jar`, а `jars-cp` содержит все другие JAR-файлы приложений и зависимостей, то можно запустить *ServiceMonitor* следующим образом:

```
$ java
    --module-path jars-mp
    --class-path 'jars-cp/*'
    --module monitor/monitor.Main
```

Вы можете увидеть полученный модульный граф на рис. 8.6. Некоторые детали могут быть неясны (например, почему все три автоматических модуля попали в граф, даже если в командной строке был указан только *monitor*?). Не переживайте; я объясню это в следующих подразделах.

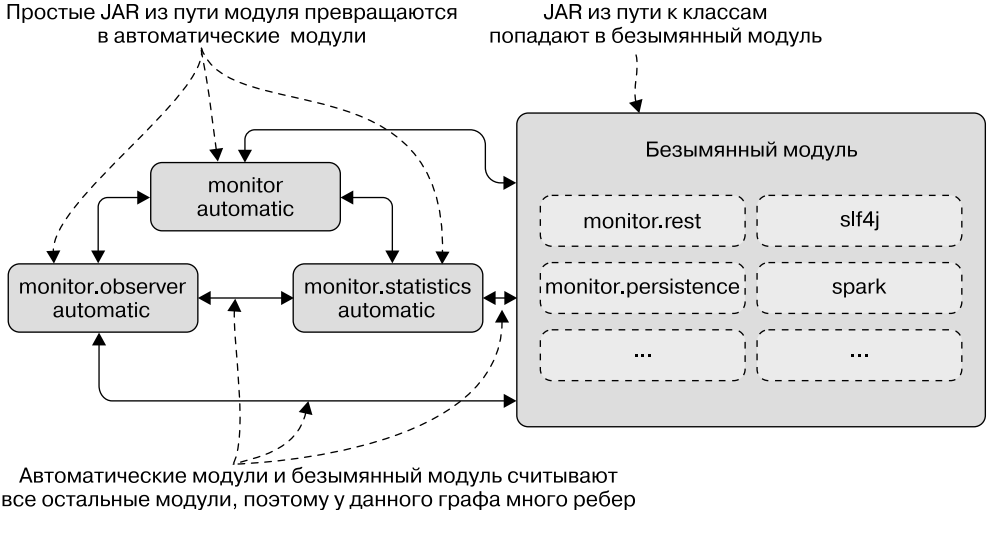


Рис. 8.6. С помощью простых JAR-файлов monitor.jar, monitor.observer.jar и monitor.statistics.jar в пути модуля JPMS создает для них три автоматических модуля. Содержимое пути класса попадает в безымянный модуль, как и раньше. Обратите внимание, как автоматические модули читают друг друга и безымянный модуль, создавая много циклов в графе

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Автоматические модули — полноценные именованные. Это значит, что:

- на них можно ссылаться по имени в декларациях других модулей, например запрашивая их;
- строгая инкапсуляция не позволяет им использовать внутренние компоненты платформенного модуля (в отличие от безымянного модуля);
- они подлежат проверке на разделенные пакеты.

С другой стороны, у них есть несколько особенностей. Я хочу обсудить их, прежде чем начать всерьез использовать автоматические модули в разделе 9.2.

8.3.1. Имена автоматических модулей: мелочь, имеющая большое значение

Главная цель превращения простых JAR-файлов в модули — это возможность запрашивать их в модульных декларациях. Для этого им нужно имя, но откуда его взять в отсутствие дескрипторов?

Сначала записи манифеста, затем имя файла

Один из способов определить простое имя модуля JAR основан на его манифесте, который представляет собой файл `MANIFEST.MF` в папке `META-INF` JAR. Манифест содержит все виды информации в виде пар «заголовок — значение». Один из наиболее важных заголовков — `MainClass`, который определяет точку входа одномодульного приложения, называя класс, содержащий метод `main`, — это позволяет запускать приложение с помощью `java -jar app.jar`.

Если JAR в пути модуля не содержит дескриптор, то система модулей выполняет двухэтапный процесс для определения имени автоматического модуля.

1. Она ищет заголовок `Automatic-Module-Name` в манифесте. Если он существует, то используется соответствующее значение в качестве имени модуля.
2. При отсутствии заголовка в манифесте система модулей генерирует имя модуля из имени файла.

Возможность выводить имя модуля из манифеста гораздо предпочтительнее, поскольку это более стабильно — подробности см. в подразделе 8.3.4.

Точные правила для вывода имени модуля из имени файла немного сложны, но детали не являются слишком важными. Вот суть:

- имена файлов JAR часто заканчиваются версией (например, `-2.0.5`). Они распознаются и игнорируются;
- каждый символ, кроме букв и цифр, превращается в точку.

Данный процесс может привести к неудачным результатам, когда получающееся имя модуля недопустимо. В качестве примера приведу инструмент манипулирования байт-кодом Byte Buddy: он опубликован в Maven Central как

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

`byte-buddy-${`*`версия`*`}.jar`, что приводит к автоматическому названию модуля `byte.buddy`. К сожалению, это недопустимо, поскольку `byte` — ключевое слово Java. (В подразделе 9.3.3 даны советы о том, как решить такие проблемы.)

Чтобы не угадывать, какое имя система модулей выбирает для данного JAR-файла, можно использовать инструмент `jar`, который поможет узнать это:

```
$ jar --describe-module --file=${jar-файл}
```

Если в JAR отсутствует дескриптор модуля, то вывод начинается следующим образом:

```
> No module descriptor found. Derived automatic module.
>
> ${имя_модуля}@${версия_модуля} automatic
> requires java.base mandated
```

`${`*`имя_модуля`*`}` — заместитель для фактического имени — как раз то, что нужно. К сожалению, по его значению нельзя сказать, откуда было выбрано имя: из записи манифеста или имени файла. Выяснить это помогут несколько вариантов.

- Извлеките манифест с помощью `jar --file ${`*`jar-файл`*`} --extract META-INF/MANIFEST.MF` и просмотрите его вручную.
- В Linux разархивируйте `-p ${`*`jar-файл`*`} META-INF/MANIFEST.MF`, чтобы вывести манифест в терминал и, таким образом, сэкономить на открытии файла.
- Переименуйте файл и снова запустите `jar --describe-module`.

Рассмотрим Guava 20.0 в качестве примера:

```
$ jar --describe-module --file guava-20.0.jar

> No module descriptor found. Derived automatic module.
>
> guava@20.0 automatic
> requires java.base mandated
# включенные пакеты опущены
```

В качестве автоматического модуля Guava 20.0 известен как *guava*. Но это универсально или связано с названием модуля? Используя `unzip`, я рассмотрел манифест:

```
Manifest-Version: 1.0
Build-Jdk: 1.7.0-google-v5
Built-By: cgdecker
Created-By: Apache Maven Bundle Plugin
[… опущены записи, связанные с OSGi …]
```

Как видите, `Automatic-Module-Name` не установлен. Переименование файла в `com.google.guava-20.0.jar` дает имя модуля *com.google.guava*.

Если использовать менее устаревшую версию Guava (например, 23.6), то получится следующий вывод:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
$ jar --describe-module --file guava-23.6-jre.jar

> No module descriptor found. Derived automatic module.
>
> com.google.common@23.6-jre automatic
> requires java.base mandated
# включенные пакеты опущены
```

Как можно понять из того факта, что выбранное имя и имя файла не совпадают, Google выбрал *com.google.common* в качестве имени модуля Guava. Проверим это с помощью `unzip`:

```
Manifest-Version: 1.0
Automatic-Module-Name: com.google.common
Build-Jdk: 1.8.0_112-google-v7
```

Вот что мы имеем в итоге: параметр `Automatic-Module-Name` установлен.

Когда нужно задавать Automatic-Module-Name

Если вы поддерживаете проект, который был опубликован, то есть его артефакты доступны через Maven Central или другой публичный репозиторий, то следует тщательно продумать, когда устанавливать `Automatic-Module-Name` в манифесте. Как я объясню в подразделе 8.3.4, это делает использование проекта с автоматическими модулями гораздо более надежным, но также дает обещание, что будущие явные модули станут заменой для текущих JAR-файлов. По сути, вы говорите: «Вот как будут выглядеть модули; я просто еще не успел их выпустить».

Тот факт, что определение автоматического имени модуля предлагает пользователям начать полагаться на артефакты вашего проекта в качестве модулей, имеет несколько важных последствий.

- Имена будущих модулей должны быть именно такими, какими вы объявляете их сейчас. (В противном случае надежная конфигурация станет доказывать пользователям, что модули отсутствуют.)
- Структура артефакта должна оставаться неизменной, поэтому нельзя перемещать поддерживаемые классы или пакеты из одного JAR в другой. (Даже без учета модулей это нерекомендуемая практика. Но с путем класса не имеет значения, какой JAR содержит класс, так что вы можете обойтись без него. С другой стороны, когда работает система модулей, происхождение класса имеет значение, поскольку доступность заставляет пользователей запрашивать правильно указанный модуль.)
- Проект работает достаточно хорошо на Java 9+. Если для этого нужны параметры командной строки или другие обходные пути, то они хорошо документированы. (В противном случае вы не можете быть уверены, что в коде не спрятаны проблемы, которые делают другие обещания спорными.)

Разумеется, разработка программного обеспечения, скажем так, «не вполне предсказуема», вследствие чего это нельзя назвать гарантиями. Но у вас должны быть

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

веские основания полагать, что вы можете выполнить данные обещания. При отсутствии пропускной способности для тестирования на Java 9+ или же при обнаружении проблем, которые делают модульность непредсказуемой, будьте с этим честны и пока не устанавливайте `Automatic-Module-Name`. Если он был установлен и все равно нужно было внести такие изменения, то основной удар придется по версии. На рис. 8.7 показан пример настройки `Automatic-Module-Name`.

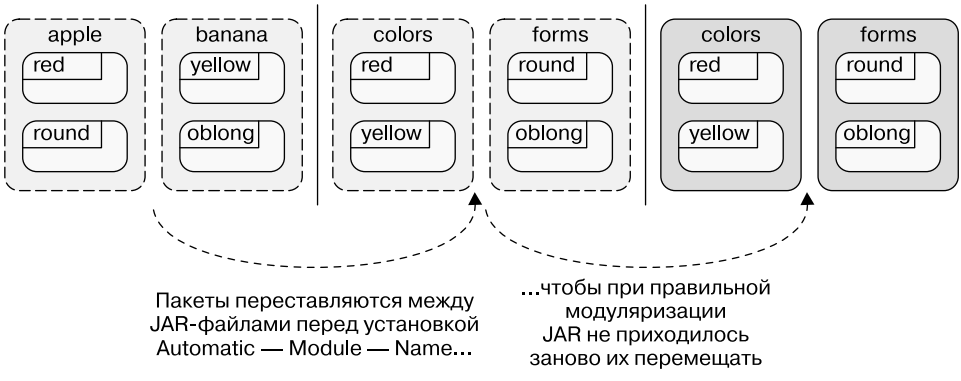


Рис. 8.7. Если вы планируете перемещать классы между пакетами или пакеты между JAR-файлами перед модульной настройкой проекта, то подождите с установкой Automatic-Module-Name до завершения настроек. Здесь JAR-файлы проекта *(слева)* были реорганизованы перед публикацией автоматического имени модуля *(в середине)*, поэтому, когда стали модульными *(справа)*, структура не изменилась

Вашему проекту не нужно ориентироваться на Java 9+, чтобы можно было установить `Automatic-Module-Name`. JAR может содержать байт-код, скомпилированный для более старых версий JVM, но определение имени модуля по-прежнему помогает пользователям, применяющим систему модулей. То же самое верно и для дескрипторов модулей, как объясняется в подразделе 9.3.4.

8.3.2. Разрешение модулей для автоматических модулей

Важный компонент для понимания и прогнозирования поведения системы модулей — понимание того, как она строит граф во время разрешения модулей. Для явных модулей это просто (необходимы директивы; см. подраздел 3.4.1); но для безымянных все немного сложнее (см. подраздел 7.2.2), поскольку простые JAR-файлы не могут выражать зависимости.

Автоматические модули также создаются из простых JAR-файлов, поэтому тоже не имеют явных зависимостей, что ставит вопрос о том, как они ведут себя во время разрешения. Мы ответим на него через мгновение, но, как вы увидите, это приводит к новому вопросу: следует ли помещать зависимости автоматического модуля в класс или путь модуля? К концу подраздела вы узнаете ответ.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Разрешение зависимостей автоматических модулей

Первый вопрос, на который нужно ответить, — что происходит во время разрешения модулей, если JPMS обнаруживает автоматический модуль. Автоматические модули были созданы для модуляризации в условиях одномодульных зависимостей, поэтому используются в ситуациях, когда разработчики активно работают над модульным представлением своего проекта. В этом случае было бы вредным, если бы автоматические модули включали почти каждый платформенный модуль (как делает безымянный модуль), поэтому они этого не делают. (Для ясности: они также не применяют какие-либо явные модули приложения.)

Тем не менее JAR имеют тенденцию зависеть друг от друга; и если система модулей разрешает лишь автоматические модули, которые были явно необходимы, то все остальные автоматические модули должны быть добавлены к графу с помощью `--add-modules`. Представьте: вы делаете это для большого проекта с сотнями зависимостей, которые помещены в путь модуля. Чтобы предотвратить такое чрезмерное и хрупкое ручное добавление модулей, JPMS загружает все автоматические модули, *как только встречает первый из них*.

Как только один автоматический модуль разрешен, то же самое происходит и со всеми остальными. Вы получаете либо все простые JAR-файлы в виде автоматических модулей (если по крайней мере один обязателен или добавлен вручную), либо ни одного (в противном случае). Это объясняет, почему на рис. 8.6 (см. выше) показаны три модуля *monitor.**, хотя лишь *monitor*, который не может выражать зависимости, был явно разрешен и сделан корневым.

Обратите внимание: автоматические модули подразумевают читабельность (см. раздел 9.1) для других автоматических модулей, то есть любой модуль, который читает *один* модуль, читает *все*. Имейте это в виду при определении зависимостей автоматических модулей: использование метода проб и ошибок может привести к меньшему количеству директив `requires`, чем необходимо.

В приложении *ServiceMonitor* модуль *monitor.rest* зависит от веб-фреймворка Spark и для примера от Guava. Обе зависимости — простые JAR-файлы, поэтому *monitor.rest* запрашивает их в качестве автоматических модулей:

```
module monitor.rest {
    requires spark.core;
    requires com.google.common;
    requires monitor.statistics;

    exports monitor.rest;
}
```

Дело в том, что одна из директив `requires` в *spark.core* или *com.google.common* может отсутствовать, и все будет продолжать работать. Как только система модулей разрешает первый автоматический модуль, она разрешает все остальные, и любой модуль, читающий один из них, читает все.

Таким образом, даже без использования `com.google.common` `guava.jar` будет выбран как автоматический модуль вместе с `spark.core.jar`; и поскольку *monitor.rest*

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

читает *spark.core*, он тоже будет читать *guava*. Обязательно правильно определите зависимости (например, с JDeps, см. приложение Г)!

Циклы в графе модулей

В словах «автоматические модули читают все остальные» скрыта деталь, на которую стоит обратить внимание: этот подход создает циклы в графах модулей. По-видимому, по крайней мере один модуль зависит от автоматического (иначе зачем бы ему там находиться?). И, таким образом, читает его, и точно так же автоматический модуль читает его.

Хоть это не имеет практического значения, я хочу пояснить, что описанное не является нарушением правила, указанного в подразделе 3.2.1, что не может быть циклов статических зависимостей. Циклы из-за автоматических модулей не объявляются статически — они вводятся динамически системой модулей.

Если автоматические модули могут читать только другие именованные модули, то все будет готово. После того как вы поместили простой JAR-файл в путь модуля, все его прямые зависимости также должны были перейти туда, а затем их зависимости и т. д., пока все переходные зависимости не станут рассматриваться как модули, явные или автоматические.

Однако превращение всех простых JAR-файлов в автоматические модули имеет свои недостатки (подробнее об этом — в подразделе 8.3.3), вследствие чего было бы неплохо иметь возможность оставить их в пути к классам и загрузить их в безымянный модуль. И система модулей предполагает именно это, позволяя автоматическим модулям читать безымянный, что означает нахождение их зависимостей в пути к классам *или* к модулям.

Выбор пути для переходных зависимостей

Как правило, у вас есть два варианта для зависимостей автоматических модулей (помните, что вы можете перечислять их с помощью JDeps): пути класса или модуля. К сожалению, не все обстоятельства позволяют вам свободно выбирать, а в некоторых случаях нужно даже сделать кое-что большее, чем просто выбрать путь.

В табл. 8.2 представлены варианты для переноса этих зависимостей в граф, исходя из того, требуются ли они другому модулю и являются ли платформенными модулями, простыми или модульными JAR. На следующих рисунках показаны конкретные ситуации:

- на рис. 8.8 демонстрируется, как платформенные модули, которые требуются только для автоматического модуля, не разрешаются по умолчанию;
- на рис. 8.9 показаны различные случаи для простых JAR-файлов, которые необходимы автоматическому модулю;
- на рис. 8.10 представлена эволюция модульного графа, если переходная зависимость превращается из простого в модульный JAR.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Таблица 8.2. Как добавить зависимости автоматического модуля в модульный граф

Зависимость требуется другому, явному модулю		
	Путь класса	**Путь модуля**
Платформенный модуль	—	✓
Простой JAR	× (зависимость не заполнена)	✓
Модульный JAR	× (зависимость не заполнена)	✓
Зависимость не требуется другому, явному модулю		
	Путь класса	**Путь модуля**
Платформенный модуль	—	! (разрешается вручную)
Простой JAR	✓	✓ (разрешается автоматически)
Модульный JAR	✓	! (разрешается вручную)

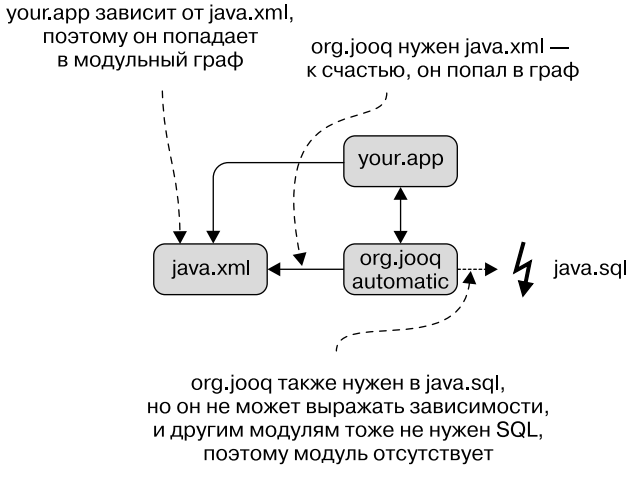


Рис. 8.8. Если в проекте (в данном случае your.app) используется автоматический модуль (org.jooq), то вы не можете быть уверены, что модульный граф построен правильно. Автоматические модули не выражают зависимости, поэтому необходимые им платформенные модули могут не попасть в граф (что и произошло здесь с java.sql) и должны быть добавлены вручную с помощью --add-modules

Сосредоточившись на платформенных модулях, мы видим, что автоматический модуль не может выражать зависимости от них. Как следствие, модульный граф может или нет содержать их; и в последнем случае автоматический модуль, скорее всего, выйдет из строя во время выполнения, выдав ошибку из-за отсутствия классов.

Единственный способ избежать этого заключается в следующем: сопровождающие проект должны публично документировать, какие модули им нужны, чтобы пользователи могли убедиться в наличии необходимых модулей. Пользователи могут сделать это, запрашивая их явно, например, в модуле, который зависит от автоматического модуля, или с помощью `--add-modules`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

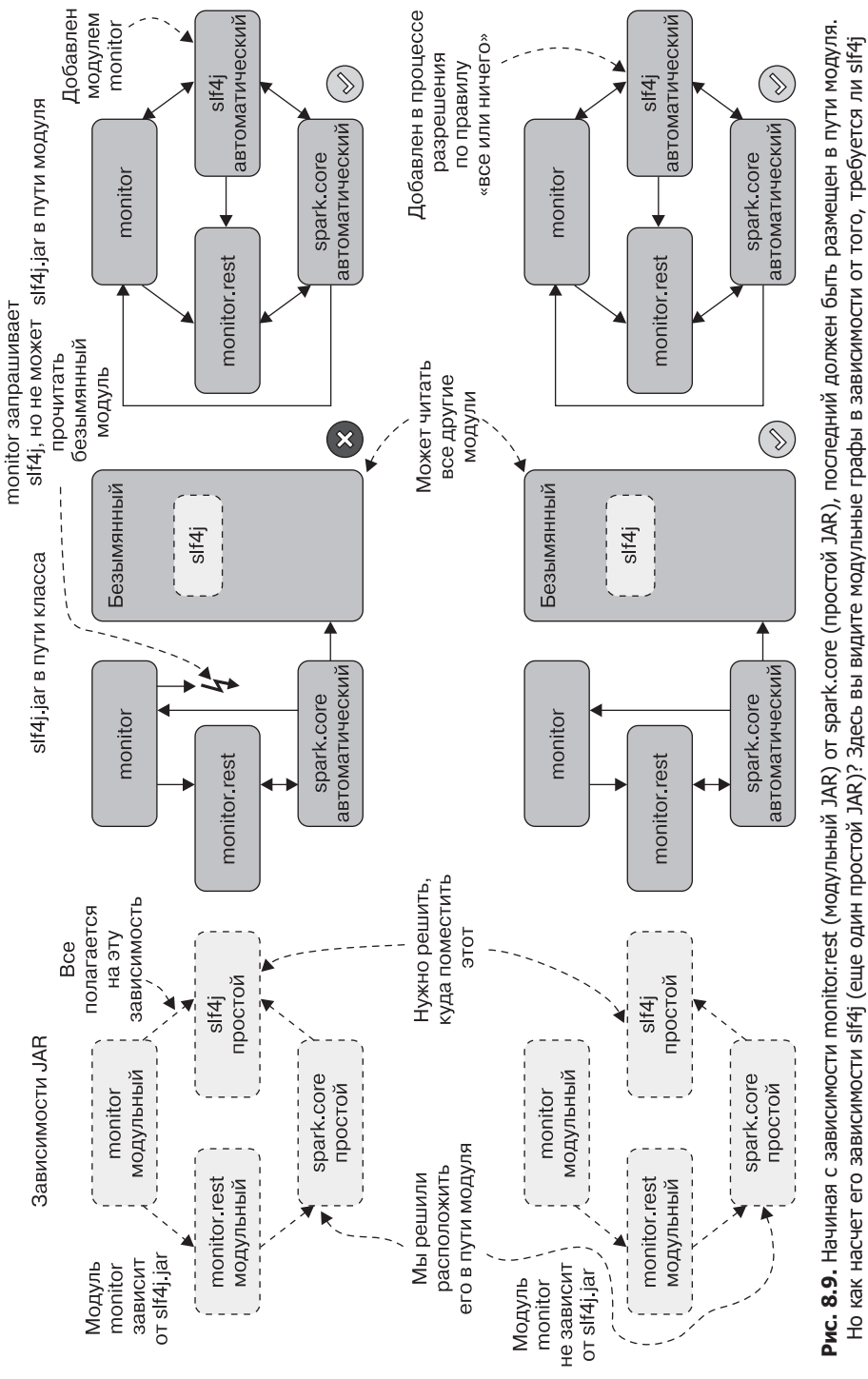


Рис. 8.9. Начиная с зависимости monitor.rest (модульный JAR) от spark.core (простой JAR), последний должен быть размещен в пути модуля. Но как насчет его зависимости slf4j (еще один простой JAR)? Здесь вы видите модульные графы в зависимости от того, требуется ли slf4j другим модульным JAR (верхний или нижний ряд) или по какому пути он расположен (средний по сравнению с правым столбцом). Выглядит так, что путь модуля очевидно выигрывает — но взгляните на рис. 8.10

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

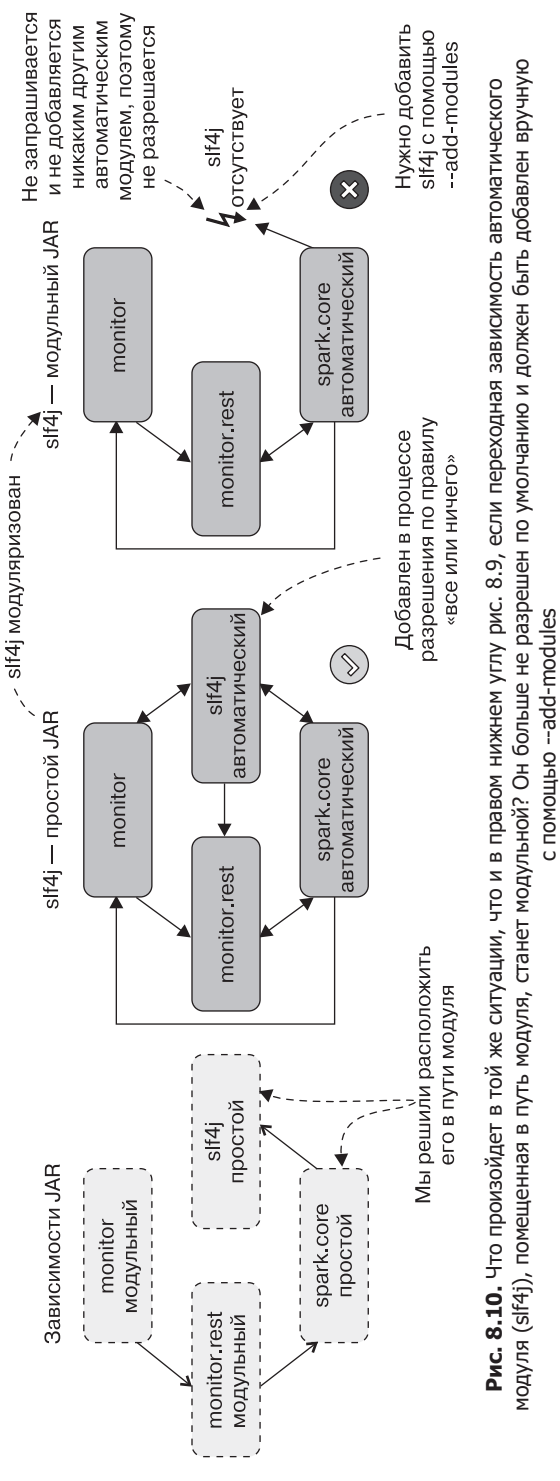


Рис. 8.10. Что произойдет в той же ситуации, что и в правом нижнем углу рис. 8.9, если переходная зависимость автоматического модуля (slf4j), помещенная в путь модуля, станет модульной? Он больше не разрешен по умолчанию и должен быть добавлен вручную с помощью --add-modules

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Отключив зависимости от платформенных модулей, посмотрим на модули приложения. Если зависимости автоматического модуля требуются явным модулям, то они должны быть помещены в путь модуля и затем разрешены системой модулей — больше ничего делать не нужно. При отсутствии запроса каким бы то ни было явным модулем JAR-файлы могут быть помещены либо в путь класса, где они будут свернуты в безымянный модуль и, следовательно, всегда доступны, либо в путь модуля, где какой-то другой механизм должен поместить их в граф:

- простые JAR-файлы вытягиваются с помощью правила «все или ничего» при загрузке автоматического модуля;
- платформенные модули и явные модули приложения не разрешены по умолчанию. Нужно либо запросить их от какого-либо другого модуля, либо добавить их вручную с помощью `--add-modules` (см. подраздел 3.4.3).

В сочетании с тем фактом, что большинство или даже все зависимости в какой-то момент перейдут от простых JAR к модульным, эти два наблюдения привлекают внимание: они подразумевают, что переходные зависимости из пути модуля работают нормально, пока являются простыми JAR, но исчезают из графа, как только становятся модульными.

Сосредоточимся на втором пункте и рассмотрим модули, к которым должен предоставляться доступ для одномодульных зависимостей. Если ни вы, ни другие модули не запрашивают их, то они не попадут в граф и зависимости не смогут получить к ним доступ. В этом случае можно либо запросить их в модульных дескрипторах (не забудьте добавить комментарий, зачем это нужно), либо добавить их с флагами командной строки во время компиляции и выполнения. В подразделах 9.2.2 и 9.2.3 кратко обсуждаются компромиссы, связанные с этим решением, в зависимости от конкретного сценария.

Дополнительным препятствием на пути могут быть типы, которые автоматический модуль предоставляет в своем публичном API. Предположим, что проект (модульный JAR) зависит от библиотеки (простой JAR) с методом, который возвращает `ImmutableList` из Guava (тоже простой JAR):

```
public ImmutableList<String> getAllTheStrings() {
   // ...
}
```

Если поместить проект и библиотеку в путь модуля, а Guava — в путь к классам, то получится модульный граф, показанный на рис. 8.11: проект (явный модуль) читает библиотеку (автоматический модуль), которая читает безымянный модуль (содержащий Guava). Если код теперь вызывает метод, который возвращает `ImmutableList`, то проверка доступности для этого типа завершится не в вашу пользу, поскольку ваш модуль не читает безымянный.

Это не совсем в новинку. Будь `ImmutableList` непубличным типом библиотеки, его нельзя было бы вызвать из-за отсутствия видимости. И так же, как в данном случае, все зависит от объявленного типа возвращаемого значения. Если бы метод вместо этого объявил, что возвращает `List`, а затем выбрал `ImmutableList` в качестве конкретного возвращаемого типа, то все было бы хорошо. Речь не о том, какой тип объявляет API, а о том, какой возвращает.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

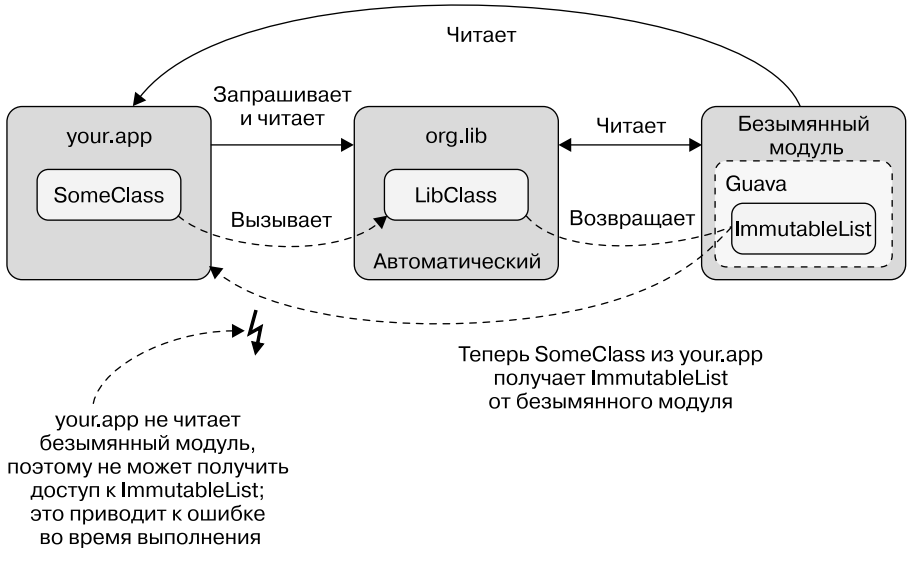


Рис. 8.11. Если метод в автоматическом модуле (в данном случае org.lib) возвращает тип из безымянного модуля (ImmutableList), то именованные модули (your.app) не могут получить к нему доступ, поскольку не читают безымянный модуль. Это приводит к сбою приложения в момент, когда метод объявляет, что возвращает недоступный тип (ImmutableList). Объявление супертипа (здесь, скорее всего, List) будет работать

Следовательно, если автоматический модуль предоставляет типы из другого JAR, то этот JAR также должен находиться в пути модуля. В противном случае его типы окажутся в безымянном модуле, где недоступны для явных модулей. Это может привести к `IllegalAccessError` из-за отсутствия ребер читабельности, как описано в подразделе 3.3.3.

Если, несмотря на все усилия, вы попадаете в ситуацию, когда именованному модулю требуется доступ к безымянному, то остается буквально один вариант. Параметр командной строки `--add-reads`, представленный в подразделе 3.4.4, может применяться для добавления ребра читабельности из именованного модуля в безымянный, путем использования `ALL-UNNAMED` в качестве целевого значения. Это тем не менее связывает модульный код с непредсказуемым содержимым пути к классу, вследствие чего его можно использовать только как последнее средство.

Благодаря использованию `--add-reads` пример с Guava в пути к классам и автоматическим модулем, возвращающим `ImmutableList`, может в итоге сработать. Если явный модуль, который получает экземпляр `ImmutableList` (и впоследствии не проходит проверку доступности), был назван *app*, то добавление `--add-reads app=ALL-UNNAMED` и для компилятора, и для среды выполнения заставило бы приложение работать.

Учитывая все вышесказанное, когда и какой путь нужно выбирать? Идти ва-банк с автоматическими модулями или предпочесть оставить как можно больше зависимостей в пути к классам? Читайте далее, чтобы узнать.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

8.3.3. Все в автоматических модулях?

Если можно размещать простые JAR-файлы в пути модулей, чтобы превратить их в автоматические модули, разве все еще нужен путь к классам? Разве нельзя поместить каждый JAR-файл в путь модуля, превратив их все в явные или автоматические модули (в зависимости от того, содержат ли они дескриптор)? Технический ответ на этот вопрос: да, можно. Тем не менее я не рекомендую это делать — позвольте объяснить почему.

Простые JAR-файлы не создают хорошие модули

Вообще говоря, простые JAR-файлы не являются хорошими модулями:

- они могут получить доступ к внутренним API JDK (см. раздел 7.1);
- они могут разделять пакеты между собой и модулями JEE (см. раздел 7.2);
- они не выражают свои зависимости.

Если они превратились в автоматические модули, то система модулей наложит на них свои правила и вам, возможно, придется потратить некоторое время на исправление возникающих проблем. Кроме того, как только простой JAR обновляется до модульного, он больше не разрешается по умолчанию (см. табл. 8.2 и рис. 8.10), поэтому для каждого такого обновления где-либо в дереве зависимостей проекта потребуется добавить его вручную. Единственное преимущество автоматических модулей — они могут запрашиваться явными модулями. Если в этом нет необходимости, то за все, сделанное автоматически, вы получите множество других проблем.

С другой стороны, если оставить путь к классу, то JAR-файлы будут свернуты в безымянный модуль, где:

- нелегальный доступ по умолчанию разрешен как минимум для еще одного выпуска Java;
- различия между JAR-файлами не имеют значения, хотя все же существуют между JAR-файлами и платформенными модулями;
- они могут читать все модули платформы Java SE, если содержат точку входа приложения;
- ничего не нужно делать, когда простой JAR обновляется до модульного.

Это значительно упрощает жизнь.

ВАЖНАЯ ИНФОРМАЦИЯ

Несмотря на восхищение от того, что в модуле есть все, я рекомендую поместить в путь модуля только минимальное количество простых JAR-файлов, необходимых для работы проекта, а остальные — в путь классов.

Модуляризованные зависимости автоматического модуля, с другой стороны, обычно должны находиться в пути модуля. Поскольку они представляют собой модульные JAR-файлы, им не нужно, чтобы система модулей относилась к ним так же

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

мягко, как и к безымянному модулю; при загрузке в виде модулей они получают надежную конфигурацию и строгую инкапсуляцию.

Автоматические модули как мост к пути классов

Существует философский взгляд на работу с меньшим количеством автоматических модулей: это превращает их в мост между модульным миром и хаотическим путем класса (рис. 8.12). Модули могут располагаться на одной стороне и запрашивать свои прямые зависимости как автоматические модули, а переходные зависимости — оставаться на другой стороне. Каждый раз, когда одна из зависимостей превращается в явный модуль, она оставляет модульную сторону моста и выводит на него свои прямые зависимости как автоматические модули. Это нисходящий подход, о котором я упоминал ранее; мы рассмотрим его более подробно при обсуждении стратегий модуляризации в разделе 9.2.

Рис. 8.12. Длинный мост Бьена в Ханое, 1939 год. Фото предоставлено пользователем manhhai. Используется в Creative Commons CC BY 2.0

8.3.4. Зависимость от автоматических модулей

Единственная цель автоматических модулей — зависеть от простых JAR-файлов, поэтому становится возможным создавать явные модули, не дожидаясь, пока все зависимости будут модульными. Однако есть важное предостережение: если манифест JAR не содержит запись `Automatic-Module-Name`, то зависимость по своей природе ненадежна.

Как объясняется в подразделе 8.3.1, без этой записи автоматическое имя модуля генерируется из имени файла. Но в зависимости от их настройки разные проекты могут задействовать разные имена для одних и тех же JAR-файлов. Кроме того,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

в большинстве проектов используется поддерживаемый Maven локальный репозиторий, где файлы JAR называются `${ID_артефакта}-${версия}`, из которых система модулей, скорее всего, выведет *${ID_артефакта}* в качестве имени автоматического модуля. Это проблематично, поскольку идентификаторы артефактов обычно не следуют обратной схеме доменных имен, определенной в подразделе 3.1.3: как только проект станет модульным, имя модуля, скорее всего, изменится.

Google Guava продолжает быть отличным примером, поскольку используется часто. Как было показано, для `guava-20.0.jar` система модулей получает автоматическое имя модуля *guava*. Это имя файла в локальном репозитории Maven, но другие проекты могут иметь другую конфигурацию.

Допустим, один из них называет JAR `${ID_группы}-${ID_артефакта}-${версия}`, и в этом случае файл станет называться `com.google.guava-guava-20.0.jar`, а автоматическим именем модуля будет *com.google.guava.guava*. Модуляризованная библиотека Guava, с другой стороны, станет именоваться *com.google.common*, поэтому ни одно из автоматических имен модуля не было правильным.

Таким образом, один и тот же JAR может получить разные имена модулей в разных проектах (в зависимости от их настройки) и в разное время (до и после модульности). Это может привести к потоку хаоса.

Подумайте о своем любимом проекте и представьте, что одна из ваших зависимостей ссылается на одну из *его* зависимостей как автоматический модуль с именем, которое не соответствует конфигурации проекта (рис. 8.13). Может быть, файлы зависимости называются `${ID_группы}-${ID_артефакта}-${версия}`, тогда как вы используете Maven и называете их `${ID_артефакта}-${версия}`. Теперь для зависимости требуется автоматический модуль `${ID_группы}.${ID_артефакта}`, но система модулей сгенерирует `${ID_артефакта}` в проекте. Это сломало бы сборку — и хотя есть способы поправить дело (см. подраздел 9.3.3), все они не из приятных.

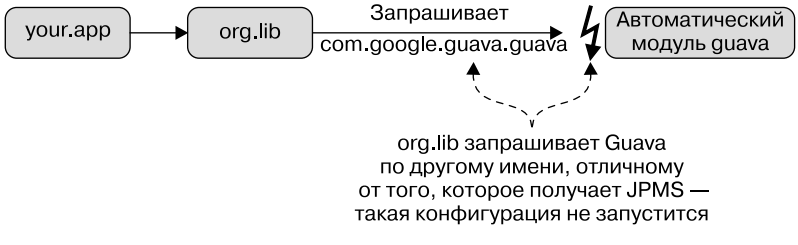


Рис. 8.13. Зависимость org.lib запрашивает Guava по имени автоматического модуля, который он получил в сборке, то есть com.google.guava.guava. К сожалению, в системе артефакт называется guava.jar, поэтому имя модуля guava является производным. Если это не исправить, то система модулей будет жаловаться на отсутствие зависимости

И все становится еще хуже! Придерживайтесь этого же проекта и мысленно добавьте еще одну зависимость, для которой требуется тот же автоматический модуль, но с другим именем (рис. 8.14). Это и есть модульный алмаз смерти, описанный в подразделе 3.2.2: один JAR не может удовлетворить требования модулей с разными именами, а несколько JAR с одинаковым содержимым не будут работать из-за

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ограничений для разделенных пакетов. Подобной ситуации нужно избегать любой ценой!

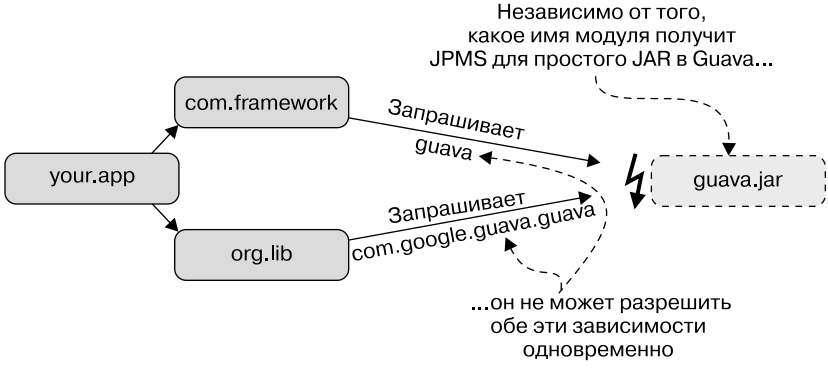


Рис. 8.14. По сравнению с рис. 8.12 ситуация ухудшилась. Еще одна зависимость, com.framework, тоже зависит от Guava, но запрашивает ее по другому имени (guava). Теперь один и тот же JAR-файл должен отображаться в виде двух модулей с разными именами, что не будет работать

В обоих случаях может показаться, что критической ошибкой было запрашивать простой JAR-файл по имени модуля, основанному на имени его файла. Но это не так — использование данного подхода подойдет для приложений и в других сценариях, где разработчик имеет полный контроль над дескрипторами модулей, запрашивающими такие автоматические модули.

Последняя капля, переполнившая чашу, заключалась в *публикации* модулей с такими зависимостями в публичном репозитории. Только тогда пользователи могут оказаться в ситуации, когда модуль неявно зависит от деталей, которые они не контролируют, и это может повлечь дополнительную работу или даже неразрешимые расхождения.

Вывод заключается в том, что *никогда* нельзя публиковать (в публичном репозитории) модули, для которых требуется простой JAR-файл без записи `Automatic-Module-Name` в его манифесте. Только с этой записью автоматические имена модулей достаточно стабильны, чтобы на них можно было положиться.

Да, вышеописанное может означать, что нельзя будет опубликовать модульную версию библиотеки или фреймворка, пока в зависимости не будет добавлена данная запись. Это прискорбно, но если так не сделать, то вы окажете пользователям медвежью услугу.

ПРИМЕЧАНИЕ

Миграция и модуляризация — мы рассмотрели все проблемы и механизмы, которые применяются к существующим кодовым базам. Перейдите к главе 9, чтобы узнать, как их лучше всего применять. После этого в части III описываются более сложные функции системы модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Резюме

- Наращиваемая модуляризация часто использует пути класса *и* модуля. Важно понимать, что любой JAR в пути класса, будь то простой или модульный, превращается в безымянный модуль и любой JAR в пути модуля превращается в именованный модуль — либо автоматический (для простого JAR), либо явный (для модульного JAR). Это позволяет пользователю JAR (а не его создателю) определить, становится ли данный модуль именованным.
- Безымянный модуль — это особенность совместимости, которая позволяет системе модулей работать с путем класса.
  - Он захватывает содержимое пути класса, не имеет имени, читает все остальные модули, а также экспортирует и открывает все пакеты.
  - Поскольку у него нет имени, явные модули не могут ссылаться на него в своих декларациях. Одно из следствий этого — то, что они не способны прочитать безымянный модуль и, следовательно, никогда не могут использовать типы, которые определены в пути к классам.
  - Если безымянный модуль является начальным, то для обеспечения корректного набора модулей используется определенный набор правил. По большому счету это не модули JEE и их зависимости. Данное действие позволяет коду из пути к классам читать все API Java SE без дальнейшей настройки, что обеспечивает максимальную совместимость.
- Автоматические модули — особенность миграции, которая позволяет модулям зависеть от простых файлов JAR.
  - Для каждого файла JAR из пути модуля создается автоматический модуль. Его имя определяется заголовком `Automatic-Module-Name` в манифесте JAR (если он имеется) или иным образом из его имени файла. Он читает все остальные модули, включая безымянный, и экспортирует и открывает все пакеты.
  - Это обычный именованный модуль, и на него можно ссылаться в декларациях модуля, например запрашивать его. Это позволяет проектам, которые являются модульными, зависеть от других, которые таковыми еще не стали.
  - Зависимость автоматического модуля может быть помещена в пути класса или модуля. Какой из них использовать, зависит от обстоятельств, но разумное значение по умолчанию — размещение модульных зависимостей в путь модуля, а простых — в путь класса.
  - Как только будет разрешен первый автоматический модуль, то же произойдет со всеми остальными. Кроме того, любой модуль, который читает один автоматический модуль, читает их все из-за подразумеваемой читабельности. Примите это во внимание при тестировании зависимостей от автоматических модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

9 Стратегии миграции и модуляризации

В этой главе

- Подготовка к переходу на Java 9 и выше.
- Непрерывная интеграция изменений.
- Постепенная модуляризация проектов.
- Генерация декларации модуля с помощью JDeps.
- Взлом сторонних JAR-файлов с помощью инструмента jar.
- Публикация модульных JAR для Java 8 и старше.

В главах 6, 7 и 8 обсуждаются технические подробности перехода на Java 9+ и превращения существующей кодовой базы в модульную. В этой главе мы более подробно рассмотрим, как наилучшим образом объединить эти подробности в успешные усилия по миграции и модуляризации. Сначала обсудим, как провести постепенную миграцию, которая хорошо согласуется с процессом разработки, особенно с инструментами сборки и непрерывной интеграцией. Далее рассмотрим, как использовать безымянный модуль и автоматические модули в качестве строительных блоков для конкретных стратегий модуляризации. И наконец, разберем варианты превращения JAR в модульные — их самих или их зависимостей. К концу этой главы вы не только поймете механизмы, стоящие за миграционными проблемами, но и узнаете, как наилучшим образом использовать их в своих целях.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

9.1. Стратегии миграции

Благодаря знаниям, полученным в главах 6 и 7, вы стали готовыми к любому бою, в который Java 9+ может втянуть вас. Теперь пришло время расширить кругозор и разработать более масштабную стратегию. Как лучше организовать фрагменты, чтобы сделать миграцию максимально тщательной и предсказуемой? В этом разделе приведены рекомендации по подготовке к миграции, оценке усилий по ее проведению, настройке непрерывной сборки на Java 9+ и преодолению недостатков параметров командной строки.

ПРИМЕЧАНИЕ

Многие темы в данном разделе связаны с инструментами сборки, но они достаточно универсальны, поэтому вам не требуется знание какого-либо конкретного инструмента. В то же время я хотел поделиться своим опытом с Maven (единственным инструментом сборки, который я использовал на Java 9+), поэтому иногда упоминал функцию Maven, применяемую мной для выполнения определенных требований. Я не буду вдаваться в подробности, поэтому вам самим придется выяснить, как работают все эти функции.

9.1.1. Подготовительные обновления

Итак, если вы еще не используете Java 8, то советую тотчас обновить приложение до этой версии! Сделайте себе одолжение и не переходите сразу на две или более версии Java. Обновитесь, приведите в действие все инструменты и процессы, на некоторое время запустите приложение, а затем приступайте к следующему обновлению. Те же действия подойдут, если нужно обновиться с Java 8 до 11, — делайте это по одному шагу за раз. Если возникнут какие-либо проблемы, то вы *действительно* захотите узнать, какая версия Java или обновление зависимостей вызвали их.

Говоря о зависимостях, еще одно действие, которое можно сделать, даже не принимая во внимание Java 9+, — это начать обновлять и их, и инструменты. Помимо общих преимуществ обновления, можно непреднамеренно обновить версию, в которой есть проблемы с Java 9+, до версии, прекрасно с ней совместимой. Вы даже не заметите, что проблема существовала. Если версии, совместимой с Java 9+, еще нет, то обновление зависимости или инструмента по-прежнему упростит процесс перехода после публикации совместимой версии.

Обзор охвата с AdoptOpenJDK

AdoptOpenJDK, «сообщество членов групп пользователей, разработчиков и поставщиков Java, которые являются сторонниками OpenJDK», предоставляет список различных проектов с открытым исходным кодом и их совместимостью с последней и следующей версиями Java: mng.bz/90HA.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

9.1.2. Оценка усилий

Есть пара вещей, которые можно сделать, чтобы понять предстоящее, и мы рассмотрим их в первую очередь. Следующим шагом будет оценка и классификация найденных проблем. Я завершу этот подраздел небольшой заметкой об оценке конкретных чисел.

Поиск проблем

Вот наиболее очевидный порядок действий для составления списка проблем.

- Настройте сборку для компиляции и тестирования на Java 9+ (Maven: набор инструментов) в идеале таким образом, чтобы можно было собирать все ошибки вместо того, чтобы останавливаться на первой (Maven: `--fail-never`).
- Запустите всю сборку на Java 9+ (Maven: `~/.mavenrc`), снова собирая все ошибки.
- Если вы разрабатываете приложение, то создайте его как обычно (имеется в виду еще не в Java 9+), а затем запустите в Java 9+. Используйте `--illegal-access=debug` или `deny` для получения дополнительной информации о несанкционированном доступе.

Тщательно проанализируйте вывод, запишите новые предупреждения и ошибки и попытайтесь связать их с тем, что обсуждалось в предыдущих главах. Обратите внимание на удаленные параметры командной строки, описанные в подразделе 6.5.3.

Рекомендуется применить несколько быстрых исправлений, таких как добавление экспорта или модулей JEE. Это позволит заметить более сложные проблемы, которые могут скрываться за доброкачественными. На данном этапе ни одно исправление не является слишком быстрым или слишком грязным — все, что заставляет сборку выдать новую ошибку, считается победой. Если выводится слишком много ошибок компиляции, то можно скомпилировать проект на Java 8 и просто запустить тесты на Java 9+ (Maven: `mvn surefire:test`).

Затем запустите JDeps для проекта *и зависимостей*. Проанализируйте зависимости от внутренних API JDK (см. подраздел 7.1.2) и запишите все модули JEE (раздел 6.1). Кроме того, обратите внимание на разделение пакетов между платформенными модулями и JAR-файлами приложения (см. подраздел 7.2.5).

Наконец, найдите в кодовой базе вызовы `AccessibleObject::setAccessible` (см. подраздел 7.1.4), приведения к `TOURLClassLoader` (раздел 6.2), анализ системных свойств `java.version` (см. подраздел 6.5.1) или напрямую вписанные в код URL ресурсов (см. раздел 6.3). Поместите все, что нашли, в один большой список — теперь пришло время проанализировать его.

Насколько все плохо?

Найденные проблемы должны быть разделены на две категории: «Я видел подобное в этой книге» и «Что, & *! #?, происходит?». Далее разделите проблемы из первой категории на «Имеет по крайней мере временное решение» и «Это сложная проблема». Особо сложные проблемы — удаленные API и разделения пакетов между платформенными модулями и JAR, которые не реализуют утвержденный стандарт или самостоятельную технологию.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Важно не путать распространенность с важностью! Вы можете получить около тысячи ошибок, поскольку отсутствует модуль JEE, но это можно с легкостью исправить. С другой стороны, у вас большие проблемы, если основная функция зависит от приведения загрузчика классов приложения к `URLClassLoader`. Или же может присутствовать критическая зависимость от удаленного API, но, поскольку вы хорошо спроектировали свою систему, это просто вызывает несколько ошибок компиляции в одном подпроекте.

Хороший подход состоит в том, чтобы задавать себе вопрос о каждой конкретной проблеме, для которой нет известного решения: «Насколько было бы плохо, если бы я вырезал проблемный код и все, что от него зависит?» Насколько это повредит проекту? Таким образом, можно ли временно деактивировать проблемный код? Тесты можно игнорировать, а функции — переключать с помощью флагов. Узнайте, насколько реально отложить исправление и запустить сборку и приложение без него.

К концу анализа у вас должен получиться список проблем в этих трех категориях:

- известная проблема с легким решением;
- известная сложная проблема;
- неизвестная проблема, требующая изучения.

Для проблем в последних двух категориях нужно знать, насколько они опасны для проекта и насколько легко можно обойтись без их неотложного исправления.

Об измерении оценки

Скорее всего, кто-то захочет, чтобы вы дали оценку, которая включает в себя несколько точных цифр — например, в часах или валюте. По умолчанию это уже сложно, но в данном случае оценивание особенно проблематично.

Миграция к Java 9+ ставит вас лицом к лицу с музыкой давно принятых решений. Проект может быть тесно связан с устаревшей версией веб-фреймворка, который нужно было обновить еще несколько лет назад, или же он накопил большой технический долг вокруг несопровождаемой библиотеки. И, к сожалению, оба из них перестают работать в Java 9+. Что нужно сделать сейчас — это погасить некоторые технические долги, и, как всем известно, оценить пени и проценты будет весьма непросто. Наконец, точно так же, как в старой доброй битве с боссом, критическая проблема — та, которая стоит дороже всех, — может быть скрыта за несколькими другими нарушителями спокойствия, поэтому ее нельзя увидеть, не подобравшись вплотную. Я не говорю, что эти сценарии очень *вероятны*, просто они *возможны*, поэтому будьте осторожны, гадая, сколько времени может уйти на переход к Java 9.

9.1.3. Непрерывная интеграция в Java 9+

Учитывая предположение, что вы постоянно разрабатываете проект, следующим шагом будет создание успешной сборки Java 9+. Есть много решений, требующих принятия.

- Какую ветку нужно собрать?
- Нужна ли отдельная версия?

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- Как разделить сборку, если она не может полностью работать на Java 9+ с первого дня?
- Как поддерживать параллельные сборки Java 8 и Java 9+?

В конце концов, нужно найти ответы, соответствующие вашему проекту и настройке непрерывной интеграции (НИ). Позвольте поделиться идеями, которые хорошо работали во время моей собственной миграции, — вы можете объединить их так, как вам нравится.

Какую ветку собирать

У вас может возникнуть соблазн настроить собственную ветку для миграции и позволить НИ-серверу создать данную ветку на Java 9+, а остальные — на Java 8, как и раньше. Но миграция может занять некоторое время, что приведет к долгоживущей ветке, и я обычно стараюсь не делать это по разным причинам:

- вы сами по себе и ваши изменения не находитесь под постоянным контролем команды, которая работает на их основе;
- обе ветки могут претерпеть множество изменений, что увеличивает вероятность конфликтов при обновлении или объединении ветки Java 9+;
- если для того, чтобы продлить изменения в основной ветке разработки к ветке Java 9+, потребуется некоторое время, то остальная часть команды тогда же может свободно добавлять в нее код, который создаст новые проблемы в Java 9+ при отсутствии немедленной обратной связи.

Хотя проведение первоначального исследования миграции в отдельной ветке может иметь смысл, я рекомендую сначала переключиться на основную ветку разработки и настроить НИ там. Это требует немного больше усилий для инструмента сборки, поскольку понадобится разделить некоторые части конфигурации (например, параметры командной строки для компилятора) по версии Java (компилятору Java не нравятся неизвестные параметры).

Какую версию собирать

Должна ли сборка Java 9+ создавать отдельную версию артефактов — что-то вроде `-JAVA-LATEST-SNAPSHOT`? Вслед за решением создать отдельную ветку Java 9+, вероятно, придется создавать отдельную версию. В противном случае можно легко смешать артефакты снимков файловой системы из разных веток, что нарушает сборку тем чаще, чем больше веток отклоняется. Если вы решили проводить сборку из основной ветки разработки, то создание отдельной версии может быть непростым, но я никогда не пытался это делать, поскольку не нашел на то веских причин.

Независимо от способа работы с версиями, при попытке заставить что-то работать на Java 9+ вы, вероятно, иногда будете собирать один и тот же подпроект с одной и той же версией на Java 8. Единственное, что я делаю снова и снова, даже если решаю не делать, — устанавливаю артефакты, собранные с помощью Java 9+, в локальный репозиторий. Знаете, тот самый рефлекторный `mvn clean install`?

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Это не очень хорошая идея: тогда нельзя будет использовать эти артефакты в сборке Java 8, поскольку она не поддерживает байт-код Java 9+.

При локальной сборке на Java 9+ старайтесь не устанавливать артефакты! Я для этого использую `mvn clean`.

Что собирать на Java 9+

Конечная цель — запустить инструмент сборки на Java 9+ и собрать все проекты на всех этапах/задачах. В зависимости от того, сколько элементов в данном списке вы создали ранее, возможно, потребуется всего лишь изменить парочку вещей, чтобы этого достичь. В таком случае выполните данные действия — не нужно усложнять процесс. С другой стороны, если список более сложный, то разделить сборку Java 9 можно несколькими способами:

- запустить сборку на Java 8 и только компилировать и тестировать на Java 9+. Я поговорю об этом чуть позже;
- провести миграцию для каждой цели/задачи. Это значит, что сначала нужно попытаться скомпилировать весь проект с помощью Java 9+, прежде чем запускать тесты;
- провести миграцию по подпроекту, то есть сначала попытаться скомпилировать, протестировать и упаковать один подпроект, прежде чем переходить к следующему.

Вообще говоря, я предпочитаю подход «по целям/задачам» для крупных монолитных проектов и подход «по подпроектам», если проект разбит на части, достаточно маленькие, чтобы их можно было разрешить за один раз.

Если вы выбираете подпроект, но по какой-либо причине один из подпроектов не может быть собран на Java 9+, то не получится легко собрать подпроекты, зависящие от него. Однажды я попал в такую ситуацию, и мы решили настроить сборку Java 9 в два этапа.

1. Собрать все на Java 8.
2. Собрать все на Java 9+, кроме проблемных подпроектов (подпроекты, зависящие от них, были собраны на основе артефактов Java 8).

Инструмент сборки для Java 9+

Пока проект полностью не переведен на Java 9+, вам, возможно, придется часто переключаться между сборками для 8 и 9+. Посмотрите, как можно настроить версию Java для выбранного инструмента сборки, не устанавливая версию Java по умолчанию для всей машины (Maven: `~/.mavenrc` или набор инструментов). Затем подумайте об автоматизации переключения. В этой ситуации я написал небольшой сценарий, который установил `$JAVA_HOME` в JDK 8 и JDK 9+, чтобы можно было быстро выбрать именно то, что нужно.

И еще — инструмент сборки может не работать должным образом на Java 9+. Вероятно, он нуждается в модуле JEE или плагин использует удаленные API. (У меня была такая проблема с плагином JAXB для Maven, которому нужен был *java.xml.bind*, поскольку плагин полагается на его внутренние компоненты.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В этом случае стоит рассмотреть возможность запуска сборки на Java 8 и только компиляции или тестирования на Java 9+, но данный способ не сработает, если сборка делает что-то с созданным байт-кодом (для Java 9+) в собственном процессе (который на Java 8). (Я столкнулся с этой проблемой с компилятором вызова удаленных методов Java (`rmic`); он заставил нас запустить всю сборку на Java 9+, хотя мы бы предпочли этого не делать.)

Приняв решение запустить сборку на Java 9+, даже если она не очень хорошо работает, придется настроить процесс сборки с некоторыми новыми параметрами командной строки. Делать это так, чтобы вашим коллегам по команде было легко (никто не хочет добавлять опции вручную), в то же время поддерживая его на Java 8 (который не знает новых опций), может быть весьма непросто (Maven: `jvm.config`). Я не нашел способа заставить его работать на обеих версиях без переименования файла, поэтому пришлось включить его в сценарий «Переключить версию Java».

Как настроить сборку Java 9+

Как сохранить работоспособность сборок Java 8 *и* Java 9+, когда необходимо добавить параметры конфигурации, зависящие от версии, в компилятор, тестовую среду выполнения или в другие задачи? Инструмент сборки должен помочь. Скорее всего, он имеет функцию, которая позволяет адаптировать общую конфигурацию к различным обстоятельствам (Maven: профили). Ознакомьтесь с этим, поскольку, возможно, в конечном итоге придется часто его использовать.

При работе с параметрами командной строки для JVM, зависящими от версии, существует альтернатива, позволяющая инструменту сборки разобрать их: с помощью нестандартного параметра JVM `-XX:+IgnoreUnrecognizedVMOptions` можно дать виртуальной машине указание игнорировать неизвестные параметры командной строки во время запуска (данный параметр недоступен в компиляторе). Несмотря на то что он позволяет использовать одни и те же параметры как для Java 8, так и для Java 9+, я не рекомендую делать это по умолчанию, поскольку будут отключены некоторые проверки — а они могут помочь найти ошибки. Взамен я предпочитаю разделять параметры по версии, если только это возможно.

Тестирование в обоих направлениях

Если вы работаете с библиотекой или фреймворком, то не можете контролировать путь, путь класса или путь модуля, где пользователи размещают исходный JAR-файл. В зависимости от проекта это может иметь значение, и тогда необходимо тестировать оба варианта.

К сожалению, здесь я не могу дать никаких советов. На момент написания книги ни Maven, ни Gradle не имели хорошей поддержки для запуска тестов один раз в каждом пути, и вам может потребоваться создание второй конфигурации сборки. Будем надеяться, что поддержка инструментов со временем улучшится.

Сначала исправить, потом решить

Как правило, большинство элементов в списке проблем Java 9+ легко исправить с помощью флага командной строки. Например, легко экспортировать внутренний API.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Однако это не решает основную проблему. Иногда решение такое же простое, скажем, замена внутреннего `sun.reflect.generics.reflectiveObjects.NotImplementedException` на `UnsupportedOperationException` (я не шучу: мне приходилось делать это не раз), но чаще всего это не так.

Стоит стремиться к быстрым и грязным исправлениям или к правильным решениям, которые занимают немного больше времени? На этапе создания полноценной сборки я рекомендую быстрые исправления:

- при необходимости добавьте флаги командной строки;
- деактивируйте тесты, желательно только на Java 9+ (на JUnit 4 для этого легко использовать предположения; на JUnit 5 я рекомендую условия);
- переключите подпроект обратно на компиляцию или тестирование в Java 8, если он использует удаленный API;
- если ничего не помогает, то полностью пропустите проект.

Работающая сборка, которая дает немедленную обратную связь всей команде о совместимости их проекта с Java 9+, стоит многого, в том числе быстрого доступа к ней. Чтобы иметь возможность улучшить эти временные исправления позже, я рекомендую создать систему, которая поспособствует их идентификации.

Я отмечаю временные исправления комментарием, например, `////[JAVA НЕДАВНЕЕ, <ПРОБЛЕМА>]:<пояснение>`, поэтому полнотекстовый поиск по `JAVA НЕДАВНЕЕ, GEOTOOLS` приводит меня ко всем тестам, которые ранее пришлось отключить, поскольку версия GeoTools не была совместима с Java 9.

Обычно обнаруживаются новые проблемы, которые изначально были скрыты за предыдущей ошибкой сборки. Если это произошло, то обязательно добавьте их в список проблем Java 9+. Точно так же вычеркните уже решенные.

Держим индикаторы зелеными

После настройки успешной сборки должна нарисоваться полная картина всех проблем Java 9+, с которыми вы сталкиваетесь. Настало время решить их одну за другой.

Некие проблемы могут быть трудными или требующими времени для решения; вы даже можете определить, что их нельзя рассмотреть до самого конца — возможно, вплоть до выпуска важной версии или до появления в бюджете небольшого места для маневра. Не волнуйтесь, если это займет какое-то время. Со сборкой, которую каждый разработчик в команде может сломать и исправить, вы никогда не сможете сделать шаг в неправильном направлении; даже если впереди много работы, вы в конечном итоге доберетесь до финиша небольшими шагами.

9.1.4. Мысли о параметрах командной строки

В Java 9+ можно в конечном итоге использовать больше параметров командной строки, чем когда-либо прежде, — говорю о своем собственном опыте. У меня есть парочка идей, о которых я хочу рассказать далее:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- четыре способа применения параметров командной строки;
- зависимость от слабой инкапсуляции;
- подводные камни параметров командной строки.

Пройдемся по ним по очереди.

Четыре способа применить параметры командной строки

Наиболее очевидный способ применить параметры командной строки — это использовать командную строку и добавить параметры после `java` или `javac`. Догадывались ли вы, что есть еще три возможности?

Если приложение поставляется в виде исполняемого файла JAR, то использование командной строки не подойдет. В этом случае можно задействовать новые записи манифеста `Add-Exports` и `AddOpens`, которые берут разделенный запятыми список пар `${`*`модуль`*`}/${`*`пакет`*`}` и экспортируют или открывают этот пакет для кода из пути к классам. JVM сканирует только исполняемый JAR приложения, то есть тот, который указан с опцией `-jar` среды выполнения, для этих записей манифеста, поэтому нет смысла добавлять их в JAR-файлы библиотеки.

Другой способ статически определить параметры командной строки, по крайней мере для JVM, — переменная среды `JDK_JAVA_OPTIONS`. Она была представлена в Java 9+, поэтому Java 8 ее не поддерживает. Таким образом можно добавлять любые параметры командной строки, специфичные для Java 9+, которые будут применяться при каждом выполнении `java` на компьютере. Вряд ли это пригодится в долгосрочной перспективе, но такой способ может облегчить те или иные эксперименты.

Наконец, параметры командной строки не обязательно вводить непосредственно в командной строке. Альтернативой являются так называемые *файлы аргументов* (или *@-файлы*) — текстовые файлы, на которые можно ссылаться в командной строке с помощью `@${`*`имя_файла`*`}`. Компилятор и среда выполнения будут действовать так, как если бы содержимое файла было добавлено в команду.

В подразделе 7.2.4 показано, как скомпилировать код, который использует аннотации из JEE и JSR 305:

```
$ javac
    --add-modules java.xml.ws.annotation
    --patch-module java.xml.ws.annotation=jsr305-3.0.2.jar
    --class-path 'libs/*'
    -d classes/monitor.rest
    ${исходные_файлы}
```

Здесь добавлены `--add-modules` и `--patch-module`, чтобы компиляция работала в Java 9+. Вы можете поместить эти две строки в файл с именем `java-LATEST-args` и затем скомпилировать его следующим образом:

```
$ javac
    @java-LATEST-args
    --class-path 'libs/*'
    -d classes/monitor.rest
    ${исходные_файлы}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Новое в Java 9+ — то, что JVM также распознает файлы аргументов, поэтому они могут использоваться совместно и для компиляции, и для выполнения.

> **Maven и файлы аргументов**
>
> К сожалению, файлы аргументов не поддерживаются Maven. Плагин компилятора сам создает файл для всех собственных опций, а Java не поддерживает вложенные файлы аргументов.

Зависимость от слабой инкапсуляции

Как подробно объясняется в разделе 7.1, среды выполнения Java 9–11 (или выше) по умолчанию разрешают несанкционированный доступ, выдавая только предупреждение. Эта возможность отлично подходит для запуска неподготовленных приложений, но я советую не полагаться на нее во время правильной сборки, поскольку она позволяет незаметно пропускать новые несанкционированные обращения. Вместо этого я собираю все необходимые параметры `--add-exports` и `--add-opens`, а затем активирую строгую инкапсуляцию во время выполнения с помощью `--illegal-access=deny`.

Ловушки параметров командной строки

Использование параметров командной строки чревато несколькими ловушками.

- Эти параметры заразительны в том смысле, что если нужны JAR, то нужны и всем его зависимостям.
- Мы надеемся, что разработчики библиотек и сред, которым требуются определенные параметры, сообщат, что их клиентам нужно их применять, — но никто не читает документацию, пока не станет слишком поздно.
- Разработчики приложений должны сопровождать список параметров, объединяющих требования нескольких библиотек и фреймворков, которые они используют.
- Нелегко сопровождать параметры таким образом, чтобы их можно было разделить между различными этапами сборки и выполнения.
- Сложно определить, какие параметры можно удалить из-за обновления версии, совместимой с Java 9.
- Может быть трудно применить параметры к нужным процессам Java: например, для плагина инструмента сборки, который не запускается в том же процессе, что и инструмент сборки.

Приведенные ловушки проясняют один момент: параметры командной строки — исправление, а не правильное решение, и за это придется заплатить в дальнейшем. И не случайно — они были разработаны для того, чтобы сделать нежелательное возможным. Конечно, использовать их не так просто — иначе не было бы стимула решать основную проблему.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Делайте все возможное, чтобы полагаться только на общедоступные и поддерживаемые API, не разделять пакеты и вообще избегать проблем, описанных в данной главе. И, что важно, поощряйте библиотеки и фреймворки, которые делают то же самое! Но дорога в ад вымощена благими намерениями, поэтому если ничего не помогает, то используйте каждый флаг командной строки, имеющийся в вашем распоряжении.

9.2. Стратегии модуляризации

В главе 8 вы узнали все о безымянном модуле, автоматических модулях и смешивании простых JAR-файлов, модульных JAR-файлов, путей классов и модулей. Но как вы применяете полученные знания на практике? Каковы лучшие стратегии для поэтапной модуляризации кодовой базы? Чтобы ответить на эти вопросы, представьте всю экосистему Java как огромный многоуровневый граф артефактов (рис. 9.1).

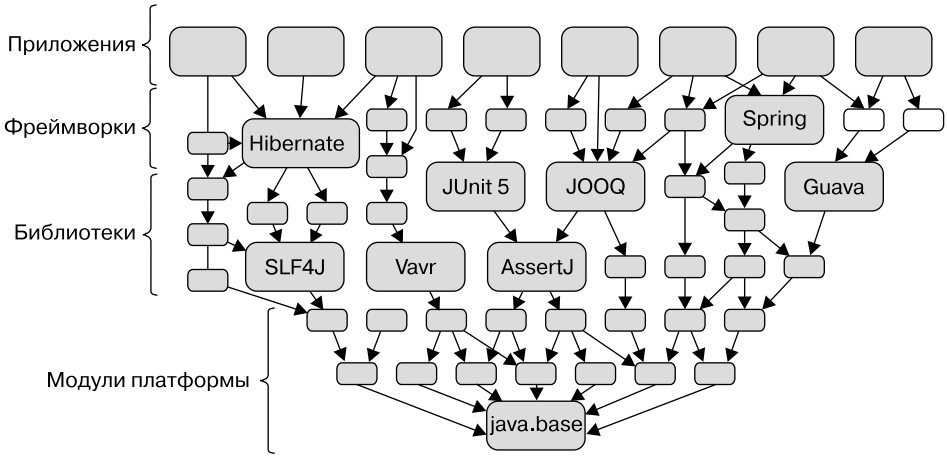


Рис. 9.1. Художественная интерпретация графа глобальных зависимостей экосистемы Java: java.base с остальной частью JDK внизу, затем библиотеки без сторонних зависимостей, выше более сложные библиотеки и структуры и приложения сверху (не обращайте внимания на какие-либо индивидуальные зависимости)

Внизу находится JDK, который раньше был одним узлом, но благодаря системе модулей теперь состоит примерно из 100 узлов с базой *java.base*. Поверх них находятся библиотеки, не имеющие зависимостей времени выполнения вне JDK (такие как SLF4J, Vavr и AssertJ), за ними следуют библиотеки с несколькими зависимостями (например, Guava, JOOQ и JUnit 5). Где-то посередине находятся фреймворки с более глубокими стеками (например, Spring и Hibernate), а на самом верху — приложения.

За исключением JDK, все эти артефакты были простыми JAR, когда вышел Java 9, и потребуется несколько лет, чтобы большинство из них стало содержать дескриптор модуля. Но как это произойдет? Как экосистема может претерпеть такие масштабные изменения, не развалившись? Стратегии модуляризации, которые появились

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

благодаря безымянному модулю (см. раздел 8.2) и автоматическим модулям (см. раздел 8.3), — вот ответ. Они позволяют сообществу Java модуляризовать экосистему практически независимо один от другого.

Разработчики, которым проще всего поддерживать проект, не имеющий зависимостей вне JDK, или чьи зависимости уже были модульными, могут реализовать *восходящую* стратегию (см. подраздел 9.2.1). Для приложений предлагается *нисходящий* подход (см. подраздел 9.2.2). Разработчикам, сопровождающим библиотеки и фреймворки с немодулированными зависимостями, достался самый сложный путь — *изнутри наружу* (см. подраздел 9.2.3).

При рассмотрении экосистемы в целом место вашего проекта в ней определяет, какую стратегию вы должны использовать. Рисунок 9.2 поможет выбрать правильный путь. Но, как объясняется в подразделе 9.2.4, эти подходы также работают в отдельных проектах, и в данном случае можно выбрать любой из трех. Однако прежде, чем перейти к этому, мы упростим изучение стратегий, предполагая, что все артефакты модуляризуются одновременно.

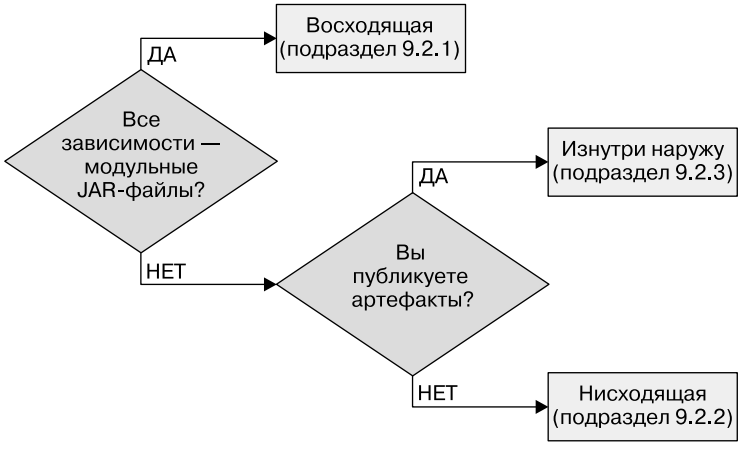


Рис. 9.2. Как определить, какая стратегия модуляризации подходит вашему проекту

Включив дескриптор модуля в JAR-файлы, вы объявляете, что проект готов к применению в качестве модуля в Java 9+. Это следует делать только в том случае, если вы предприняли все возможные шаги для обеспечения его бесперебойной работы, — главы 6 и 7 объясняют большинство проблем, но при использовании в коде рефлексии следует прочитать еще и главу 12.

Если пользователям нужно предпринять какие-то действия, чтобы модули работали, например добавить флаги командной строки в свое приложение, это должно быть хорошо задокументировано. Обратите внимание: можно создавать модульные JAR-файлы, которые по-прежнему бесперебойно работают в Java 8 и более ранних версиях, — подраздел 9.3.4 описывает эту возможность.

Как я часто упоминал, модуль имеет три основных свойства: имя, четко определенный API и явные зависимости. При создании модуля, очевидно, нужно выбрать имя. Экспорт может быть затруднен, но в основном предопределяется тем, какие

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

классы должны быть доступны. Реальные проблемы и то, где остальная часть экосистемы вступает в игру, начинаются с зависимостей. Этот раздел посвящен данному аспекту.

> **Знайте свои зависимости**
>
> Вы должны знать совсем немного о зависимостях, прямых и косвенных, чтобы модуляризовать проект. Помните: можно использовать JDeps для определения зависимостей (в частности, для модулей платформы; см. приложение Г) и `jar --describe-module` для проверки статуса модуляризации JAR (см. подразделы 4.5.2 и 8.3.1).

Учитывая все вышесказанное, пришло время посмотреть, как работают три стратегии модуляризации.

9.2.1. Восходящая модуляризация: когда все зависимости — модульные

Это самый простой случай для преобразования JAR-файлов проекта в модули: предполагается, что код зависит только от явных модулей (прямо или косвенно). Неважно, являются ли они модулями платформы либо приложения; вы можете пойти напрямую.

1. Создайте декларации модулей, которые запрашивают все ваши прямые зависимости.
2. Поместите JAR-файлы с зависимостями, отличными от JDK, в путь модуля.

Теперь вы полностью модульно реализовали свой проект — мои поздравления! Если вы сопровождаете библиотеку или фреймворк и пользователи размещают JAR-файлы проекта в пути модуля, то те станут явными модулями и пользователи могут начать получать преимущества системы модулей. На рис. 9.3 показан пример восходящей модуляризации.

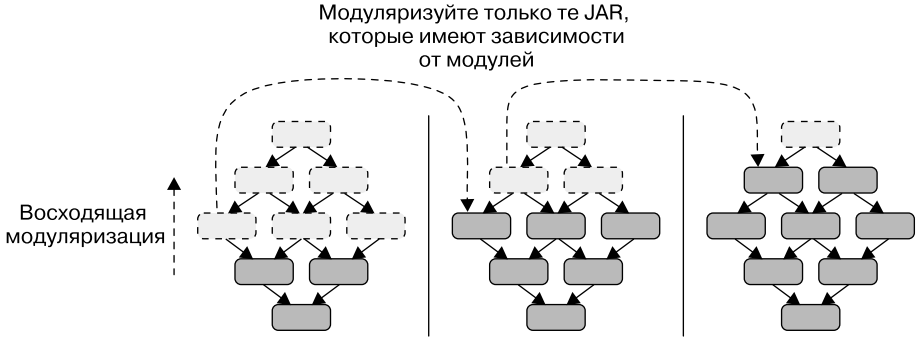


Рис. 9.3. Артефакты, зависящие от модульных JAR-файлов, могут сразу же быть модульными, что приводит к восходящей миграции

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Почти так же важно, но менее очевидно благодаря тому, что все JAR-файлы из пути к классам попадают в безымянный модуль (см. раздел 8.2), его не обязательно использовать в качестве модуля. Если кто-то еще придерживается пути класса, то проект будет работать так, словно там не было дескриптора модуля. Если вы хотите модуляризовать свою библиотеку, но зависимости проекта еще не являются модулями, то см. подраздел 9.2.3.

9.2.2. Нисходящая модуляризация: когда нет времени дожидаться всех зависимостей

Допустим, вы, как разработчик приложения, хотите в ближайшее время модуляризовать проект. Маловероятно, что все зависимости уже поставляют модульные JAR-файлы. Если они это сделают, то вам повезет и вы сможете использовать нисходящий подход, который я только что описал. В противном случае нужно будет задействовать автоматические модули и начать смешивать путь модуля и путь класса следующим образом.

1. Создайте декларации модулей, которые запрашивают все ваши прямые зависимости.
2. Поместите все модульные JAR-файлы, созданные вами, и зависимости в путь модуля.
3. Поместите все простые JAR-файлы, которые непосредственно необходимы модульным JAR-файлам, в путь модуля, где они превратятся в автоматические модули.
4. Подумайте, что делать с оставшимися простыми JAR-файлами (см. подраздел 8.3.3).

Возможно, проще всего будет поместить все оставшиеся файлы JAR в путь модуля в инструменте сборки или IDE и попробовать запустить. Я не думаю, что в целом это лучший подход, но в вашем случае он может сработать. Если так и произошло, то пользуйтесь.

При наличии проблем с разделением пакетов или доступом к внутренним API JDK можно попробовать поместить эти JAR-файлы в путь к классам. Потому что они нужны только автоматическим модулям и могут читать безымянный модуль, а это прекрасно работает.

В будущем, после того как ранее автоматический модуль модуляризуется, эта настройка может не сработать, поскольку теперь тот является модульным JAR в пути модуля и, следовательно, не может получить доступ к коду из пути к классам. Я считаю, это хорошо, так как позволяет лучше понять, какие зависимости являются модулями, а какие — нет. Это также хорошая возможность проверить дескриптор модуля и узнать больше о проекте. Чтобы устранить проблему, переместите зависимости данного модуля в путь модуля. На рис. 9.4 приведен пример нисходящей модуляризации.

Обратите внимание: не нужно беспокоиться о происхождении автоматических имен модулей (см. подраздел 8.3.4). Однако если они основаны на имени файла, то нужно будет изменить некоторые директивы `requires`, когда они получат явное имя модуля; но, поскольку вы контролируете все декларации модулей, это не так важно.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

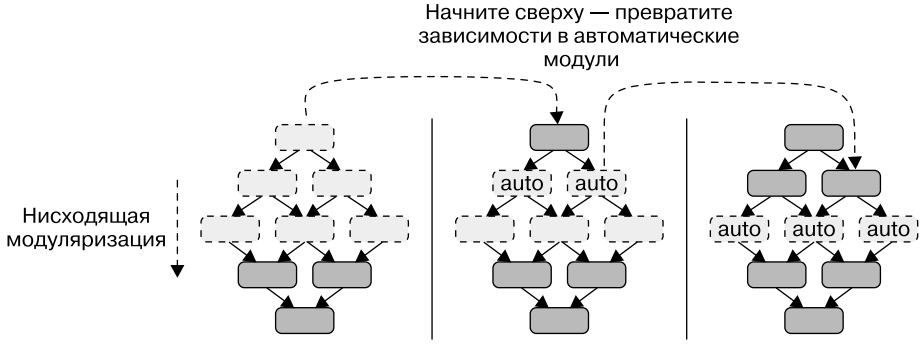


Рис. 9.4. Благодаря автоматическим модулям становится возможным модулировать артефакты, которые зависят от простых JAR-файлов. Приложения могут использовать это для нисходящей модуляризации

Как насчет того, чтобы модули, которые требуются для немодульных зависимостей, попали в граф? Приложение может потребовать их в модульной декларации или использовать `--add-modules` с целью добавить их вручную во время компиляции и запуска. Последний вариант доступен только в том случае, если вы управляете командой запуска. Инструмент сборки может принимать эти решения, но вам все равно нужно знать о параметрах и о том, как их настраивать, чтобы можно было исправлять проблемы по мере их возникновения.

9.2.3. Модуляризация изнутри наружу: если проект находится в середине стека

Большинство библиотек и, в частности, фреймворки не находятся ни внизу, ни вверху стека — что им делать? Они модуляризуются изнутри наружу. В этом процессе есть немного от восходящего подхода (см. подраздел 9.2.1), поскольку выпуск модульных JAR-файлов не заставляет пользователей применять их в качестве модулей. Кроме этого, он работает и как при нисходящем подходе (см. подраздел 9.2.2), с одним важным отличием: планируется *публиковать* созданные модульные JAR. На рис. 9.5 показан пример модуляризации изнутри наружу.

Как я подробно описал в подразделе 8.3.4, публикация модулей с зависимостями от автоматических модулей должна происходить только в том случае, если эти простые JAR-файлы определяют запись `Automatic-Module-Name` в своем манифесте. В противном случае риск возникновения проблем в будущем при изменении имени модуля слишком велик.

Описанное может означать, что вы еще не можете модуляризовать проект. Если вы находитесь в такой ситуации, то не поддавайтесь искушению сделать это в любом случае, иначе у пользователей могут возникнуть сложные проблемы.

Я хочу сделать еще один шаг: изучить прямые и косвенные зависимости и убедиться, что ни одна из них не зависит от автоматического модуля, имя которого получено из имени JAR-файла. Ищите любую зависимость, которая не является модульным JAR и не определяет запись `Automatic-Module-Name`. Я не стал бы

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

публиковать артефакт с дескриптором модуля, извлекающим *любой* такой JAR-файл, — будь то моя зависимость или чья-либо еще.

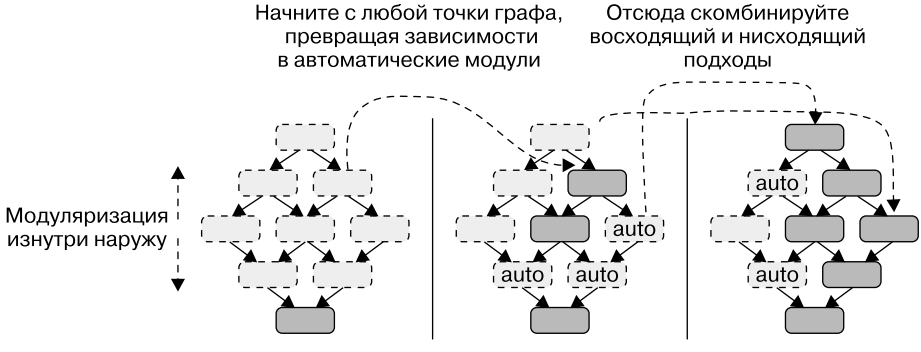


Рис. 9.5. При осторожном использовании автоматических модулей библиотеки и фреймворки в середине стека могут публиковать модульные JAR-файлы, даже если их зависимости и пользователи могут быть простыми JAR-файлами, таким образом модулируя экосистему изнутри наружу

Существует также тонкое различие, когда речь идет о платформенных модулях, которые нужны немодульным зависимостям, но не вам. Обратите внимание: приложения могут легко использовать параметры командной строки, а библиотеки или фреймворки — нет. Можно лишь задокументировать только для пользователей, что параметры должны быть добавлены, но некоторые пользователи могут упустить это из виду. Поэтому я советую явно запрашивать все платформенные модули, которые нужны немодульным зависимостям.

9.2.4. Применение данных стратегий к проекту

То, какая именно из трех стратегий должна использоваться, определяется местом проекта в гигантском, общесистемном графе зависимостей. Но если проект довольно большой, то вы не сможете сразу модуляризовать его в полном объеме, и вам может быть интересно, как сделать это постепенно. Хорошая новость: можно применять аналогичные стратегии в меньших масштабах.

Часто проще всего применить восходящую стратегию к проекту, сначала модуляризуя подпроекты, которые зависят только от кода вне вашей кодовой базы. Это работает особенно хорошо, если зависимости уже являются модульными, но не ограничиваются данным сценарием. Если это не так, то нужно применить нисходящую логику к нижней ступени подпроектов, заставив их использовать автоматические модули, чтобы они зависели от простых JAR-файлов, а затем собрать их оттуда.

Применительно к одному проекту нисходящий подход работает так же, как и применительно к экосистеме в целом. Модуляризуйте артефакт в верхней части графа, поместите его в путь модуля и превратите его зависимости в автоматические модули. Затем медленно продвигайтесь вниз по дереву зависимостей.

Можно даже сделать это изнутри наружу. В главе 10 представлены сервисы: отличный способ использовать систему модулей для разделения зависимостей как внутри проекта, так и между различными другими проектами. Данная причина хо-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

роша, чтобы начать модуляризовать систему где-то посередине графа зависимостей и оттуда двигаться вверх или вниз.

ВАЖНАЯ ИНФОРМАЦИЯ

Обратите внимание: какой бы подход ни выбрали, вы все равно не должны публиковать явные модули, зависящие от автоматических модулей, имена которых не определены в имени файла JAR, в отличие от записи Automatic-Module-Name в манифесте.

Хотя все из вышеперечисленного возможно, не следует без необходимости усложнять ситуацию. Определившись с подходом, постарайтесь быстро и методично модуляризовать проект. Разработка этого процесса и создание модулей тут и там означает, что будет сложно разобраться с графом зависимостей проекта, — и это является противоположностью одной из важных целей системы модулей: надежной конфигурации.

9.3. Делаем JAR-файлы модульными

Чтобы превратить простой JAR в модульный JAR, нужно всего лишь добавить декларацию модуля в исходный код. Легко, правда? Да (подождите), но (а вот и загвоздка!) об этом шаге можно сказать больше, чем кажется на первый взгляд.

- Возможно, вы захотите создать открытые модули (в подразделе 9.3.1 можно получить краткое объяснение).
- Вы можете быть перегружены созданием десятков или даже сотен деклараций модулей и потому ожидаете инструмента, который выполнит всю грязную работу (см. подраздел 9.3.2).
- Возможно, вы захотите модуляризовать JAR, который сами не создавали, или, вероятно, зависимость запутала дескриптор модуля и вам нужно это исправить (см. подраздел 9.3.3).
- Вы можете задаться вопросом о дескрипторах модулей в JAR, созданных для Java 8 или более ранних версий: реально ли вообще это сделать (см. подраздел 9.3.4)?

В данном разделе рассматриваются все эти темы, чтобы вы могли убедиться: ваши вложения приведут к максимальной отдаче.

9.3.1. Открытые модули как промежуточный шаг

При непрерывной модуляризации приложения может быть полезна концепция *открытых модулей*. Она подробно рассматривается в подразделе 12.2.4, но суть состоит в том, что открытый модуль отказывается от строгой инкапсуляции: все его пакеты экспортируются и открываются для рефлексии. Это значит, что все его публичные типы доступны во время компиляции, а все остальные типы и члены доступны через рефлексию. Он создается путем добавления `open module` в начало модульной декларации.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Открытые модули пригодятся, когда вы недовольны макетом пакета JAR. Может быть, пакетов слишком много или многие из них содержат публичные типы, которые вы бы предпочли не иметь, — в обоих случаях рефакторинг может занять слишком много времени. Или, может быть, модуль интенсивно используется в рефлексии и вы не хотите проходить через определение всех пакетов, которые нужно открыть.

В таких случаях открытие всего модуля — хороший способ перенести эти проблемы в будущее. Применяются предостережения о техническом долге — данные модули отказываются от строгой инкапсуляции, которая лишает их ее преимуществ.

ВАЖНАЯ ИНФОРМАЦИЯ

Поскольку превращение открытого модуля в обычный инкапсулированный модуль — несовместимое изменение, библиотеки и фреймворки никогда не должны разрешать запуск с открытым модулем с целью закрыть его позже. Трудно придумать причину, по которой такой проект должен когда-либо публиковать открытый модуль. Лучше использовать его только для приложений.

9.3.2. Генерация модульных деклараций с помощью JDeps

Если проект большой, то может потребоваться создание десятков или даже сотен модульных деклараций, что является непростой задачей. К счастью, в большинстве случаев можно использовать JDeps, поскольку основная работа является механической:

- имя модуля часто может быть получено из имени JAR;
- зависимости проекта могут быть проанализированы путем сканирования байт-кода в границах JAR;
- экспорты обратны этому анализу, то есть все пакеты, от которых зависят другие JAR-файлы, должны быть экспортированы.

Помимо этих базовых свойств, может потребоваться небольшая тонкая настройка, чтобы обеспечить запись всех зависимостей и настроить использование сервисов (см. главу 10) или более подробных зависимостей и API (см. главу 11), но все до этого момента может генерироваться через JDeps.

- Запущенный с помощью `--generate-module-info ${`*`целевой_каталог`*`} ${`*`каталог_с_jar`*`}`, JDeps анализирует все JAR-файлы в `${`*`каталог_с_jar`*`}` и генерирует `module-info.java` для каждого из них в `${`*`целевой_каталог`*`}/${`*`имя_модуля`*`}`:
- имя модуля получено из имени файла JAR, как и для автоматических модулей (включая обращение к заголовку `Automatic-Module-Name`; см. подраздел 8.3.1);
- зависимости определяются на основе анализа зависимостей JDeps. Открытые зависимости помечаются ключевым словом `transitive` (см. раздел 11.1);
- все пакеты, содержащие типы, используемые другими JAR-файлами в анализе, экспортируются.

Когда JDeps генерирует файлы `module-info.java`, вы должны проверить и адаптировать их и переместить в правильные исходные папки, чтобы следующая сборка могла их скомпилировать и упаковать.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Снова предположив, что *ServiceMonitor* еще не модуляризован, вы можете использовать JDeps, чтобы сгенерировать декларации модулей. Для этого соберите *ServiceMonitor* и поместите его JAR-файлы вместе с зависимостями в каталог `jars`. Затем вызовите `jdeps --generate-module-info declarations jars`, и JDeps сгенерирует модульные декларации, записав их в структуру каталогов, показанную на рис. 9.6.

JDeps создает папку для каждого модуля и помещает в них модульные декларации, которые выглядят аналогично написанным вами вручную ранее. (Чтобы освежить это в памяти, вы можете найти их в листинге 2.2, но подробности здесь неважны.)

JDeps также может генерировать декларации для открытых модулей (см. подраздел 12.2.4) с помощью `--generate-open-module`. Имена модулей и директивы `requires` определяются, как и прежде; но, поскольку открытые модули не могут ничего инкапсулировать, экспорт не требуется и, следовательно, директивы `exports` не генерируются.

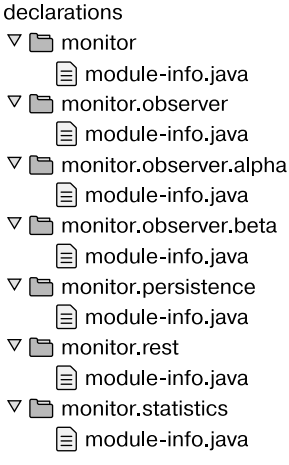


Рис. 9.6. После вызова jdeps --generate-module-info declarations jars JDeps анализирует зависимости между всеми JAR-файлами в каталоге jars (не показан) и создает для них декларации модулей в каталоге declarations (проекты, не относящиеся к ServiceMonitor, не показаны)

Проверка сгенерированных объявлений

JDeps хорошо справляется с генерацией модульных деклараций, но вы все равно должны проверять их вручную. Вам нравятся имена модулей? (Вероятно, нет, поскольку имена JAR редко следуют обратной схеме доменных имен; см. подраздел 3.1.3.) Правильно ли смоделированы зависимости? (Для получения дополнительной информации см. разделы 11.1 и 11.2.) Какие пакеты вы хотите добавить в публичный API? Может быть, вам нужно добавить парочку сервисов (см. главу 10).

Если вы разрабатываете приложение, в котором слишком много JAR-файлов, чтобы проверять все декларации вручную, и вас устраивают некоторые ошибки, то имеется более мягкий вариант: достаточно доверить поиск некритических проблем тестам, конвейеру CI, коллегам-разработчикам и тестировщикам. В этом случае убедитесь в наличии времени до следующего выпуска, чтобы удостовериться в исправлении всех неполадок.

Тем не менее при публикации артефактов необходимо очень тщательно проверять декларации! Это наиболее открытые части вашего API, и их изменение часто несовместимо — приложите все усилия, чтобы предотвратить это без веской на то причины.

Остерегайтесь отсутствующих зависимостей

Для правильной генерации JDeps директив `requires` для набора JAR-файлов все эти JAR-файлы, а также их прямые зависимости должны присутствовать в сканируемом

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

каталоге. Если зависимости отсутствуют, то JDeps сообщит об этом следующим образом:

```
> Missing dependence: .../module-info.java not generated
> Error: missing dependencies
>     depending.type -> missing.type     not found
>     ...
```

Чтобы избежать ошибочных деклараций, они не создаются для модулей, в которых присутствуют не все зависимости.

Генерируя модульные декларации для *ServiceMonitor*, я замалчивал эти сообщения. Некоторые косвенные зависимости отсутствовали предположительно потому, что Maven считал их необязательными, но это не мешало правильному созданию деклараций *ServiceMonitor*:

```
> Missing dependence:
>     declarations/jetty.servlet/module-info.java not generated
# дальнейшие сообщения журнала опущены
> Missing dependence:
>     declarations/utils/module-info.java not generated
# дальнейшие сообщения журнала опущены
> Missing dependence:
>     declarations/jetty.server/module-info.java not generated
# дальнейшие сообщения журнала опущены
> Missing dependence:
>     declarations/slf4j.api/module-info.java not generated
# дальнейшие сообщения журнала опущены
> Error: missing dependencies
>     org.eclipse.jetty.servlet.jmx.FilterMappingMBean
>         -> org.eclipse.jetty.jmx.ObjectMBean          not found
>     org.eclipse.jetty.servlet.jmx.HolderMBean
>         -> org.eclipse.jetty.jmx.ObjectMBean          not found
>     org.eclipse.jetty.servlet.jmx.ServletMappingMBean
>         -> org.eclipse.jetty.jmx.ObjectMBean          not found
>     org.eclipse.jetty.server.handler.jmx.AbstractHandlerMBean
>         -> org.eclipse.jetty.jmx.ObjectMBean          not found
>     org.eclipse.jetty.server.jmx.AbstractConnectorMBean
>         -> org.eclipse.jetty.jmx.ObjectMBean          not found
>     org.eclipse.jetty.server.jmx.ServerMBean
>         -> org.eclipse.jetty.jmx.ObjectMBean          not found
>     org.slf4j.LoggerFactory
>         -> org.slf4j.impl.StaticLoggerBinder          not found
>     org.slf4j.MDC
>         -> org.slf4j.impl.StaticMDCBinder             not found
>     org.slf4j.MarkerFactory
>         -> org.slf4j.impl.StaticMarkerBinder          not found
```

Тщательно проанализируйте экспорт

Директивы экспорта основаны исключительно на анализе того, какие типы нужны другим JAR-файлам. Это практически гарантирует, что библиотеки JAR увидят слишком мало экспортированных пакетов. Помните о данном факте при проверке вывода JDeps.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Как разработчику библиотеки или фреймворка, вам может быть неудобно публиковать артефакты, экспортирующие пакеты, которые вы считаете внутренними, в проект только потому, что они нужны нескольким модулям. Обратитесь к квалифицированному экспорту в разделе 11.3 для решения этой проблемы.

9.3.3. Взлом сторонних JAR-файлов

Иногда возникает надобность в обновлении сторонних JAR-файлов. Возможно, вам нужно, чтобы они превратились в явный модуль или по крайней мере автоматический модуль с конкретным именем. Вероятно, JAR уже является модульным, но его дескриптор неисправен или вызывает проблемы с зависимостями, которые не хотелось бы использовать. В таких случаях пришло время достать острые инструменты и приступить к работе. (Осторожно, не порежьтесь.)

Хороший пример для странных крайних случаев, которые обязательно существуют в такой большой экосистеме, как Java, — инструмент манипулирования байткодом Byte Buddy. Он опубликован в Maven Central как `byte-buddy -${`*`версия`*`}.jar`, и при попытке использования его в качестве автоматического модуля вы получите ответ от системы модулей:

```
> byte.buddy: Invalid module name: 'byte' is not a Java identifier
```

К сожалению, `byte` не является допустимым идентификатором Java, поскольку конфликтует с примитивным типом с тем же именем. Этот конкретный случай решается в Byte Buddy версии 1.7.3 и более поздних (с записью `Automatic-Module-Name`), но вы можете столкнуться с аналогичными крайними случаями и должны быть готовы к ним.

В целом не рекомендуется локально изменять опубликованные JAR-файлы, поскольку это трудно сделать надежным и самодокументированным способом. Гораздо легче, если процесс разработки включает локальное хранилище артефактов, такое как Nexus от Sonatype, к которому подключаются все разработчики. В подобном случае кто-то может создать обновленный вариант, изменить версию, чтобы сделать изменение очевидным (например, добавив `-patched`), а затем загрузить его во внутренний репозиторий.

Можно также выполнить модификацию во время сборки, и в этом случае стандартные JAR-файлы могут использоваться и редактироваться динамически по мере необходимости. Затем модификация становится частью сценария сборки.

Обратите внимание: ни в коем случае нельзя публиковать артефакты, которые зависят от измененных JAR-файлов! Пользователи не смогут легко воспроизвести изменения и останутся с нарушенной зависимостью. Это в значительной степени ограничивает следующие рекомендации для приложений.

Не обращая внимания на предостережения, посмотрим, как манипулировать сторонними JAR-файлами, если они плохо работают в существующем проекте. Я покажу, как добавлять или редактировать автоматическое имя модуля, добавлять или редактировать дескриптор модуля и добавлять классы в модули.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Добавление и редактирование автоматического имени модуля

Подходящей причиной для добавления автоматического имени модуля в JAR, за исключением сценария, в котором JPMS в противном случае не сможет получить имя, станет то, что проект уже определил имя в более новых версиях, но вы по какой-либо причине еще не можете обновить его. В этом случае редактирование JAR позволяет добавить имя, пригодное для будущего использования, в декларации модуля.

У инструмента `jar` есть опция `--update` (альтернатива — `-u`), которая позволяет модифицировать имеющийся архив Java. Вместе с параметром `--manifest=${`*`файл_манифеста`*`}` можно добавить что угодно к существующему манифесту, например, запись `Automatic-Module-Name`.

Возьмем старую версию Byte Buddy, 1.6.5, и убедимся, что она работает как автоматический модуль. Сначала создайте простой текстовый файл, скажем `manifest.txt` (можно выбрать любое имя), который будет содержать одну строку:

```
Automatic-Module-Name: net.bytebuddy
```

Затем используйте `jar`, чтобы добавить эту строку к существующему манифесту:

```
$ jar --update --file byte-buddy-1.6.5.jar --manifest=manifest.txt
```

Теперь проверим, сработало ли это:

```
$ jar --describe-module --file byte-buddy-1.6.5.jar

> No module descriptor found. Derived automatic module.
>
> net.bytebuddy@1.6.5 automatic
> requires java.base mandated
```

Отлично: ошибок нет и имя модуля выбрано по нашему желанию.

Тот же подход можно использовать для редактирования существующего автоматического имени модуля. Инструмент `jar` будет выдавать предупреждение `Duplicate name in Manifest`, однако новое значение тем не менее заменит старое.

Добавление и редактирование дескрипторов модулей

Если преобразования стороннего JAR в правильно названный автоматический модуль недостаточно или же возникли проблемы с явным модулем, то можно использовать `jar --update` для добавления или переопределения дескриптора модуля. Важный вариант использования последнего — устранение модульного алмаза смерти, описанного в подразделе 8.3.4:

```
$ jar --update --file ${jar} module-info.class
```

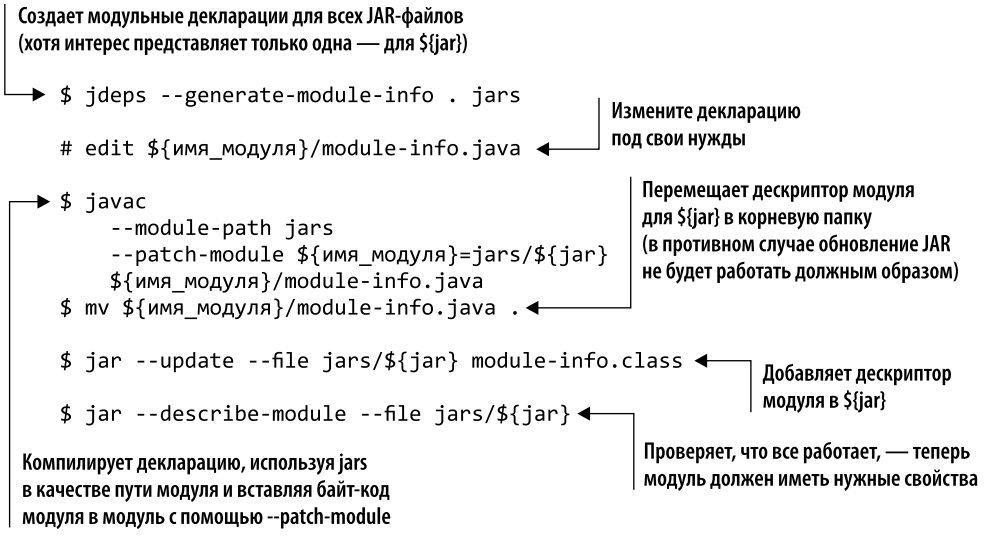
Данное действие добавляет файл `module-info.class` к `${jar}`. Обратите внимание: `--update` не выполняет никаких проверок. Это позволяет легко, случайно или преднамеренно, создавать JAR, чьи дескриптор модуля и файлы классов не согласуются, например, с необходимыми зависимостями. Пользуйтесь с осторожностью!

Более сложная задача — создать дескриптор модуля. Чтобы компилятор создал его, нужна не только декларация модуля, но и все зависимости (их наличие про-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

веряется как часть надежной конфигурации) и некоторое представление кода JAR (в виде исходников или байт-кода; в противном случае компилятор жалуется на несуществующие пакеты).

Инструмент сборки должен быть в состоянии помочь вам с зависимостями (Maven: `copydependencies`). Для кода важно, чтобы компилятор видел весь модуль, а не только декларацию. Лучше всего этого добиться, скомпилировав декларацию, в то время как байт-код модуля добавляется из его JAR с помощью `--patch-module`. В подразделе 7.2.4 представлена данная опция, а в следующем примере показано, как ее использовать:



Добавление классов к модулям

Если вам уже нужно добавить некоторые классы в пакеты зависимостей, то, возможно, вы разместили их в пути к классам. Как только эта зависимость перемещается в путь модуля, правило против разделенных пакетов запрещает такой подход. В подразделе 7.2.4 показано, как справиться с данной ситуацией динамически с помощью опции `--patch-module`. Для более долговременного решения проблемы можно еще раз использовать `jar --update`, в этом случае в целях добавления файлов классов.

9.3.4. Публикация модульных JAR-файлов для Java 8 и выше

Независимо от того, сопровождаете ли вы приложение, библиотеку или фреймворк, возможно, вы ориентируетесь на несколько версий Java. Означает ли это, что нужно уходить от системы модулей? К счастью, нет! Существует два способа представления модульных артефактов, которые отлично работают в версиях Java старше 9.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Какой бы вариант вы ни выбрали, сначала нужно создать проект для целевой версии. Можно использовать компилятор из соответствующего JDK или более новый, указав `-source` и `-target`. Если вы выбрали компилятор Java 9+, то проверьте новый флаг `--release` в разделе 4.4. Завершите этот шаг, создав JAR как обычно. Обратите внимание: данный JAR отлично работает в желаемом выпуске Java, но еще не содержит дескриптор модуля.

Следующий шаг — компиляция модульных деклараций с Java 9+. Лучший и самый надежный способ — собрать весь проект с помощью компилятора Java 9+. Теперь у вас есть два варианта, как добавить дескриптор модуля в JAR (описаны ниже).

Использование jar --update

Можно использовать `jar --update`, как описано в подразделе 9.3.3, с целью добавить дескриптор модуля в JAR. Это сработает ввиду того, что JVM до версии 9 игнорируют дескриптор модуля. Они видят только файлы других классов, и, поскольку вы собираете их для правильной версии, все просто работает.

Указанное верно для JVM, но не может обязательно относиться ко всем инструментам, обрабатывающим байт-код. Некоторые отключаются от `module-info.class` и, таким образом, становятся бесполезными для модульных JAR-файлов. Для предотвращения этого нужно будет создать мультиверсионный JAR.

Создание мультиверсионных JAR

Начиная с Java 9, `jar` позволяет создавать *мультиверсионные JAR* (MR-JAR), которые содержат байт-код для разных версий Java. Приложение Д содержит подробное введение в эту новую функцию. Чтобы извлечь максимальную пользу из данного раздела, стоит прочитать введение. Здесь я сосредоточусь на том, как использовать MR-JAR, чтобы дескриптор модуля не оказался в корне JAR.

Допустим, у вас есть обычный JAR и вы хотите превратить его в мультиверсионный, где дескриптор модуля загружается в Java 9 (и более поздние версии). Вот как это сделать с помощью `--update` и `--release`:

```
$ jar --update
    --file ${jar}
    --release 9
    module-info.class
```

Кроме того, можно создать мультиверсионный JAR за один раз:

```
$ jar --create
    --file mr.jar
    -C classes .
    --release 9
    classes-9/module-info.class
```

Первые три строки — обычный способ создания JAR из файлов классов в `classes`. Затем идет `--release 9`, за которым следуют дополнительные источники для загрузки JVM версии 9 и выше. На рис. 9.7 показан полученный JAR-файл — как вы можете заметить, корневой каталог не содержит `module-info.class`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

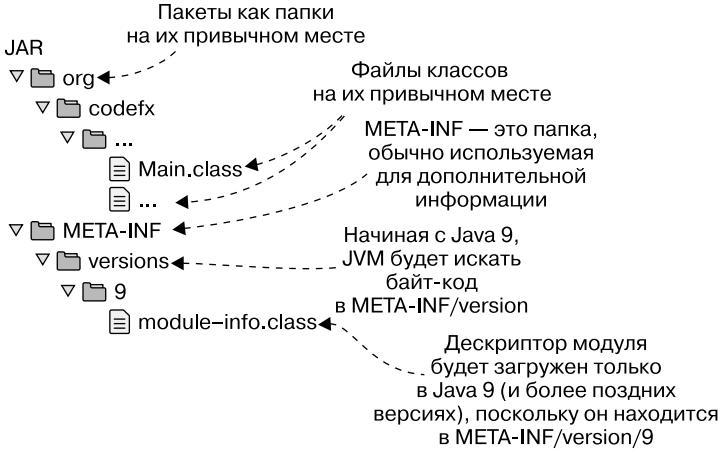


Рис. 9.7. Создав JAR с несколькими выпусками, можно будет поместить дескриптор модуля в META-INF/version/9 вместо корневого каталога артефакта

Данная функция выходит далеко за рамки добавления дескрипторов модулей. Так что я рекомендую прочитать приложение Д, если вы еще не сделали этого.

Теперь, когда мы рассмотрели основы для только что созданных проектов, а также для существующих кодовых баз, читайте дальше, чтобы узнать о расширенных функциях системы модулей в части III.

Резюме

- Если вы еще не используете Java 8, то сначала проведите обновление. В случае оповещений предварительного анализа о том, что некоторые из зависимостей вызывают проблемы в Java 9+, обновите и их. Это гарантирует, что вы выполняете все последовательно, тем самым сводя сложность к минимуму.
- Можно совершить несколько действий для анализа проблем миграции:
  - сборку на Java 9+ и применение быстрых исправлений (`--add-modules`, `--add-exports`, `--add-opens`, `--patch-module` и др.) для получения дополнительной информации;
  - использование JDeps для поиска разделенных пакетов и зависимостей от внутренних API;
  - поиск конкретных шаблонов, которые вызывают проблемы, такие как приведение к `URLClassLoader` и использование удаленных механизмов JVM.
- Собрав эту информацию, важно затем правильно ее оценить. Каковы риски быстрых исправлений? Насколько сложно их правильно разрешить? Насколько важен код для проекта?
- Начиная миграцию, постарайтесь непрерывно создавать изменения, в идеале из той же ветки, которую использует остальная команда. Это гарантирует, что усилия Java 9+ и регулярная разработка хорошо интегрированы.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- Параметры командной строки дают возможность быстро решать проблемы, с которыми вы сталкиваетесь, когда сборка работает на Java 9+, но избегайте их длительного использования. Они позволяют легко игнорировать проблемы, пока будущие выпуски Java не усугубят их. Вместо этого работайте над долгосрочным решением.
- Существует три стратегии модуляризации. Какая из них применяется к проекту в целом, зависит от его типа и зависимостей.
  - *Восходящая* — для проектов, которые зависят только от модулей. Создайте декларации модулей и поместите все зависимости в путь модуля.
  - *Нисходящая* — для приложений, чьи зависимости еще не все модульные. Они могут создавать декларации модулей и помещать все прямые зависимости в путь модуля — простые JAR-файлы превращаются в автоматические модули, от которых можно зависеть.
  - *Изнутри наружу* — для библиотек и фреймворков, чьи зависимости еще не все модульные. Работает как нисходящая, но имеет ограничение: могут применяться только автоматические модули, которые определяют запись манифеста `Automatic-Module-Name`. В противном случае автоматическое имя модуля нестабильно во время сборки, и со временем это может привести к значительным проблемам для пользователей.
  - В рамках проекта можно выбрать любую стратегию, которая соответствует его конкретной структуре.
- JDeps позволяет автоматически генерировать декларации модулей с помощью `jdeps --generate-module-info`. Это особенно актуально для крупных проектов, где написание деклараций модулей вручную занимает много времени.
- Опция `--update` инструмента `jar` позволяет изменить существующие JAR-файлы: например, установить `Automatic-Module-Name` или добавить или переписать дескриптор модуля. Если JAR-файл зависимости создает проблемы, которые невозможно исправить, то это как раз инструмент, предназначенный для их решения. Компилируя и упаковывая исходный код для более старой версии Java, а затем добавляя дескриптор модуля (либо в корневой каталог JAR, либо с помощью jar `--version` в специальный подкаталог Java 9+), можно создавать модульные JAR, которые работают в различных версиях Java и ведут себя как модуль, если размещены в пути модуля Java 9.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Часть III

Расширенные функции системы модулей

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В то время как части I и II были похожи на обеды из четырех блюд, эта часть книги больше похожа на шведский стол. Она охватывает расширенные функции системы модулей, и вы можете выбирать все, что вас больше всего интересует, в любом порядке.

Глава 10 представляет сервисы, отличный механизм для разделения пользователей и реализации API. Если вас больше интересует уточнение требований `requires` и `exports`, например, для моделирования необязательных зависимостей, то ознакомьтесь с главой 11. Изучите главу 12, чтобы подготовить модули к рефлексивному доступу с помощью вашей любимой среды и узнать, как обновить собственный рефлексивный код.

Система модулей не обрабатывает информацию о версии модуля, но вы можете записывать ее при сборке модулей и оценивать во время выполнения. В главе 13 исследуется это, а также причины, по которым больше не существует поддержки версий, что пригодилось бы для запуска нескольких версий одного и того же модуля.

Глава 14 делает шаг назад от разработки модулей и рассматривает их как входные данные для создания пользовательских образов среды выполнения, которые содержат только модули, необходимые для запуска вашего проекта. Сделав еще один шаг, вы можете добавить все приложение и создать единое развертываемое устройство для доставки его к клиентам или серверам.

Наконец, глава 15 собирает все вместе. В ней показан вариант приложения *ServiceMonitor*, использующий большинство расширенных функций, а затем дано несколько советов по проектированию и сопровождению модульных приложений. В самом конце главы я попробовал представить будущее Java: модульную экосистему.

Кстати, эти функции не являются *расширенными* в том смысле, что они сложнее основных механизмов. Дело в другом: они основаны на данных механизмах и, таким образом, требуют немного больше базовых знаний о системе модулей. Если вы прочитали часть I, особенно главу 3, то смело можете приступать к чтению части III.

(Я знаю, что уже говорил это пару раз, но помните: я сократил имена модулей, чтобы сделать их более удобными. Используйте обратную схему доменных имен, как описано в подразделе 3.1.3.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

10 Использование сервисов для разделения модулей

В этой главе

- Улучшение дизайна проектов с помощью сервисов.
- Создание сервисов, потребителей и поставщиков в JPMS.
- Использование ServiceLoader для потребления сервисов.
- Разработка хорошо продуманных сервисов.
- Развертывание сервисов в простых и модульных JAR-файлах в разных версиях Java.

До сих пор мы представляли отношения между модулями с помощью директив `requires`, где зависимый модуль должен ссылаться на каждую конкретную зависимость по имени. Как подробно объясняется в разделе 3.2, это лежит в основе надежной конфигурации. Но иногда требуется более высокий уровень абстракции.

В этой главе рассматриваются сервисы в системе модулей и их использование для разделения модулей путем удаления прямых зависимостей между ними. Первый шаг к решению любых проблем с сервисами — ознакомление с основами. После этого мы рассмотрим детали, в частности, как правильно проектировать сервисы (раздел 10.3) и применять API JDK для их использования (см. раздел 10.4). (Чтобы увидеть сервисы на практике, посмотрите ветку `feature-services` в репозитории *ServiceMonitor*.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

К концу данной главы вы узнаете, как правильно проектировать сервисы, писать декларации для модулей, которые используют или предоставляют сервисы, и загружать их во время выполнения. Вы сможете задействовать эти навыки для применения сервисов в JDK или сторонних зависимостях, а также для удаления прямых зависимостей в собственном проекте.

10.1. Изучение потребности в сервисах

Говори мы о классах, а не о модулях, нравилось бы вам всегда иметь зависимости только от конкретных типов? Или создавать полностью все зависимости для класса, который их запрашивает? Если вам нравятся шаблоны проектирования, такие как инверсия управления и внедрение зависимостей, то вы наверняка сейчас энергично покивали. Сравните листинги 10.1 и 10.2 — второй выглядит гораздо лучше, правда? Это позволяет вызывающему методу подбирать себе самый лучший поток и даже предоставляет ему свободу в выборе любой реализации `InputStream`.

Листинг 10.1. Зависит от конкретного типа и устанавливает зависимость

```
public class InputStreamAwesomizer {

        private final ByteArrayInputStream stream;

        public AwesomeInputStream(byte[] buffer) {
                stream = new ByteArrayInputStream(buffer);
        }

        // [... прекрасные методы ...]

}
```

Зависит от конкретного типа

Устанавливает зависимость напрямую

Листинг 10.2. Зависит от абстрактного типа; зависимость устанавливается вызывающим методом

```
public class InputStreamAwesomizer {

        private final InputStream stream;

        public AwesomeInputStream(InputStream stream) {
                this.stream = stream;
        }

        // [... прекрасные методы ...]

}
```

Зависит от абстрактного типа

Зависимость устанавливается вызывающим методом

Другое важное преимущество зависимости от интерфейсов или абстрактных классов и предоставления возможности кому-либо еще выбирать конкретный экземпляр — то, что это инвертирует направление зависимостей. Вместо концепций высокого уровня (скажем, `Department`), зависящих от деталей низкого уровня

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

(`Secretary`, `Clerk` и `Manager`), они оба могут зависеть от абстракции (`Employee`). Как показано на рис. 10.1, это разрушает зависимость между концепциями высокого и низкого уровня и, таким образом, разъединяет их.

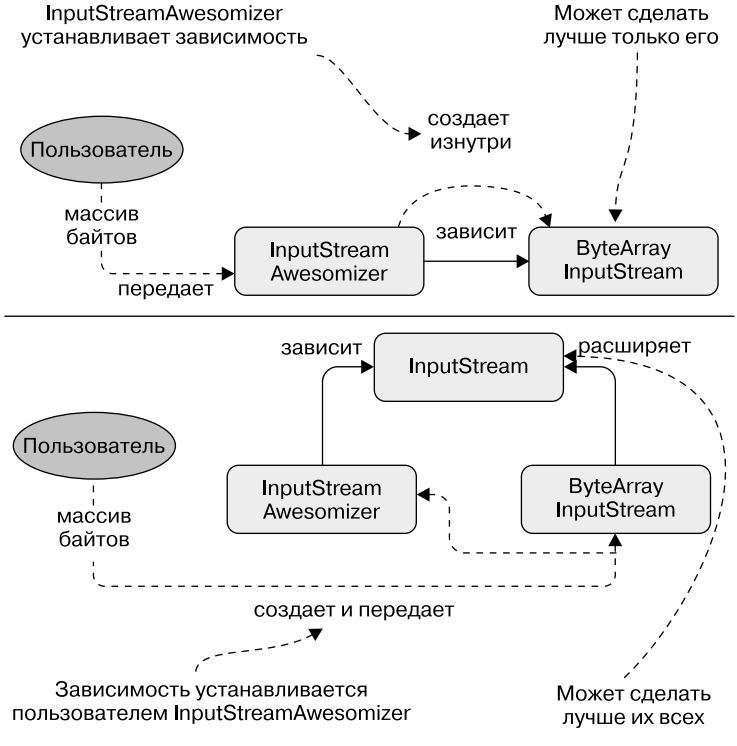


Рис. 10.1. Если тип устанавливает собственные зависимости *(сверху)*, то пользователи не могут влиять на них. В случае передачи зависимостей типа во время построения *(снизу)* пользователи могут выбрать реализацию, которая наилучшим образом подходит для их ситуации

Возвращаясь к модулям: директивы `requires` очень похожи на код в листинге 10.1, только на другом уровне абстракции:

- модули зависят от других конкретных модулей;
- пользователь не может обмениваться зависимостями;
- нет возможности инвертировать направление зависимостей.

К счастью, система модулей на этом не останавливается. Она предлагает *сервисы* — средство, с помощью которого модули могут выразить собственную зависимость от абстрактного типа или предоставить конкретный тип, удовлетворяющий ей. Система модулей располагается между ними, беря на себя роль посредника. (Если вы сейчас думаете о паттерне «Локатор сервисов», то абсолютно правы!) Как вы увидите, сервисы не могут полностью решить все упомянутые проблемы, но тем не менее имеют большое значение. На рис. 10.2 показаны два типа зависимостей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

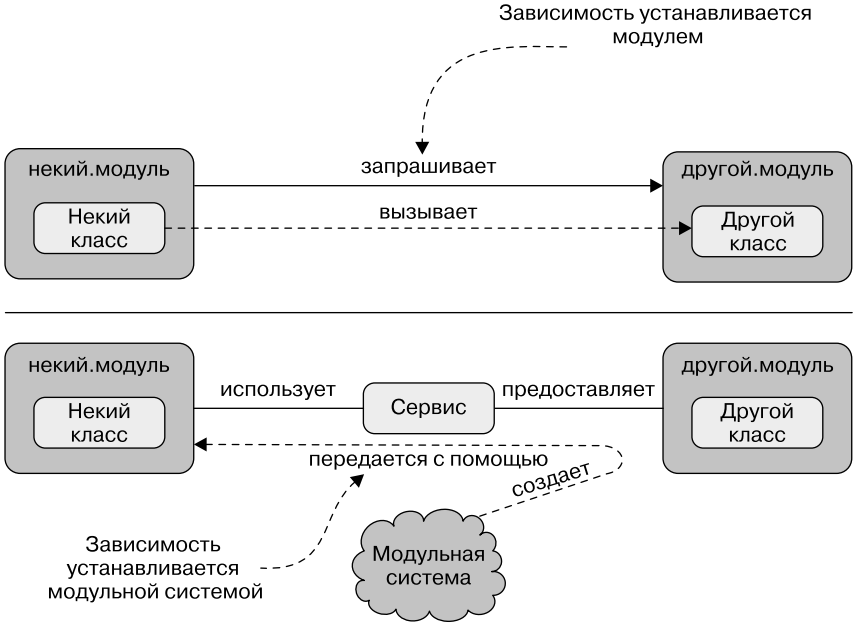


Рис. 10.2. Если модуль запрашивает другой модуль *(сверху)*, то зависимость фиксируется; она не может быть изменена извне. С другой стороны, если модуль использует сервис *(снизу)*, то конкретная реализация выбирается во время выполнения

10.2. Сервисы в модульной системе платформы Java

Когда мы говорим о сервисе в контексте JPMS, речь идет о конкретном типе, обычно интерфейсе, который мы хотим использовать, но для которого не создаем экземпляры реализаций. Вместо этого система модулей извлекает реализации из других модулей, пообещавших их предоставить, и создает экземпляры этих реализаций. В данном разделе подробно показано, как работает процесс, чтобы вы знали, какие элементы нужно поместить в модульные дескрипторы и как получать экземпляры во время выполнения, а также то, как это влияет на разрешение модулей.

10.2.1. Использование, предоставление и потребление услуг

Сервис — это доступный тип, который требуется одному модулю, а другой модуль предоставляет его экземпляр.

- Модуль, *потребляющий* сервис, выражает в нем необходимость с помощью *директивы* `users ${сервис}` в своем дескрипторе, где `${сервис}` — полное имя типа сервиса.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- Модуль, *предоставляющий* сервис, выражает предложение с помощью *директивы* `provides ${сервис}with ${поставщик}`, где `${сервис}` того же типа, что и в директиве `uses`, а `${поставщик}` — полное имя другого класса, который является одним из таких вариантов, как:
  - конкретный класс, который расширяет или реализует `${сервис}` и имеет открытый конструктор без параметров (называемый *конструктором поставщика*);
  - произвольный тип с публичным статическим методом `provide` без параметров, возвращающий тип, который расширяет или реализует `${сервис}` (называемый *методом поставщика*).

Во время выполнения зависимый модуль может использовать класс `ServiceLoader` для получения всех предоставленных реализаций сервиса путем вызова `ServiceLoader.load(${сервис}.class)`. Затем система модулей возвращает `Provider<${сервис}>` для каждого поставщика, который объявляет любой модуль в графе. На рис. 10.3 показана реализация класса `Provider`.

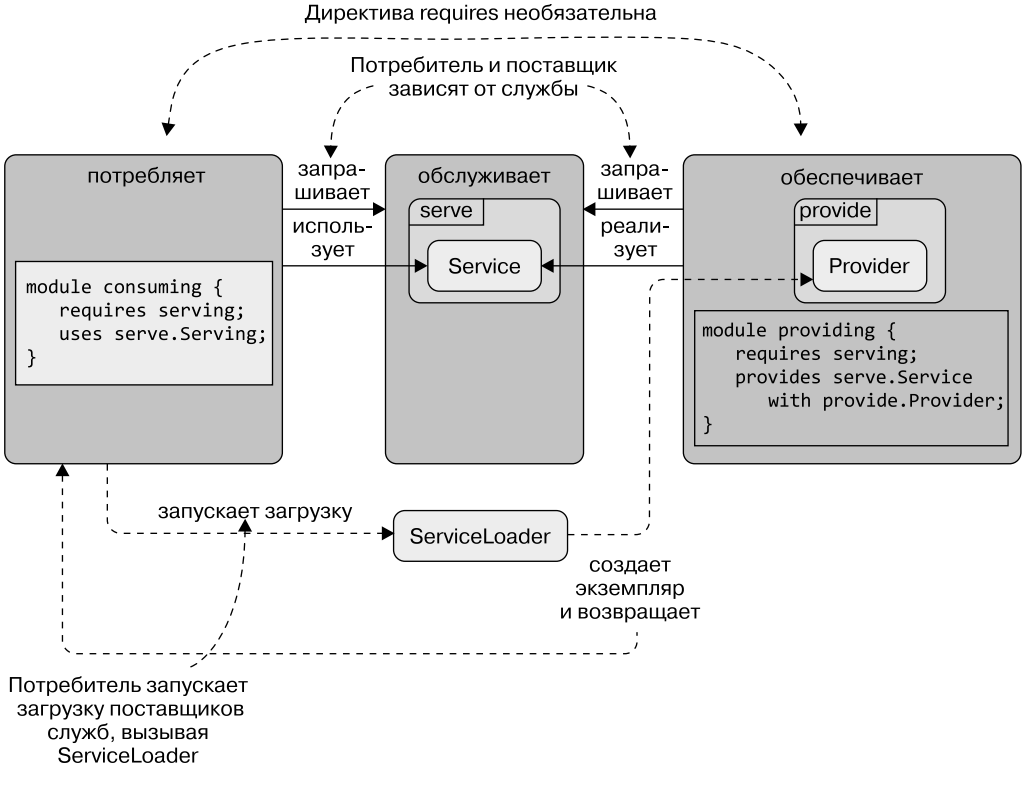


Рис. 10.3. В центре использования сервисов находится определенный тип, который здесь называется Service. Класс Provider реализует его, и модуль, в котором он находится, объявляет его с помощью директивы provides-with. Модули, применяющие сервисы, должны указать на это, прибегнув к директиве uses. Во время выполнения они затем смогут задействовать ServiceLoader для получения экземпляров всех поставщиков данного сервиса

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Существует множество деталей, которые нужно рассмотреть касательно сервисов; но, вообще говоря, они хорошо абстрагированы и просты в использовании на практике, так что начнем с этого. Устраивайтесь поудобнее; выполнение действий занимает больше времени, чем ввод директивы `requires` или `exports`.

Приложение *ServiceMonitor* представляет собой прекрасный пример хорошего использования сервисов. Классу `Monitor` из модуля *monitor* требуется `List<ServiceObserver>` для связи с сервисами, которые он должен отслеживать. Пока `Main` выполнил это следующим образом:

```
private static Optional<ServiceObserver> createObserver(String serviceName) {
    return AlphaServiceObserver.createIfAlphaService(serviceName)
        .or(() -> BetaServiceObserver.createIfBetaService(serviceName));
}
```

Не слишком важно, как именно работает код. Важно то, что он использует конкретные типы `AlphaServiceObserver` из *monitor.observer.alpha* и `BetaServiceObserver` из *monitor.observer.beta*. Следовательно, *monitor* должен зависеть от этих модулей и им необходимо экспортировать надлежащие пакеты — на рис. 10.4 показана соответствующая часть модульного графа.

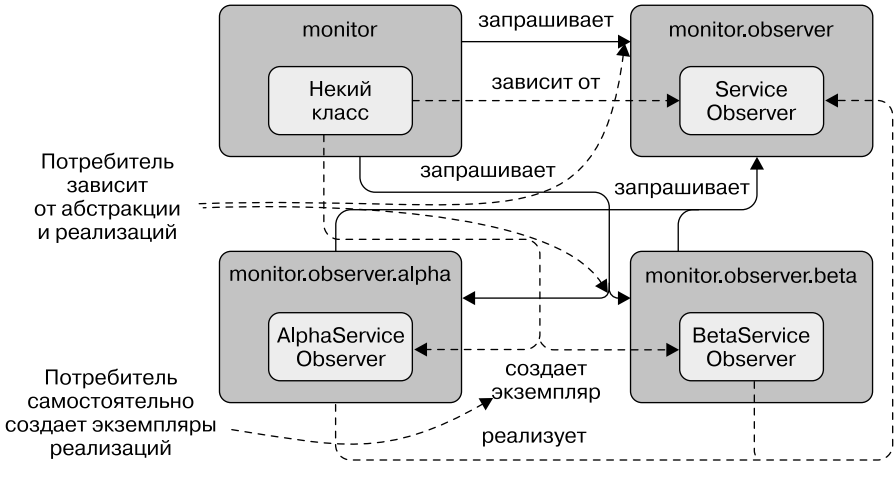


Рис. 10.4. Без сервисов модуль monitor должен зависеть от всех других задействованных модулей: observer, alpha и beta, как показано на этом частичном модульном графе

Теперь превратим это в сервисы. Итак, модуль, создающий эти `observer`, должен объявить, что он планирует использовать сервис. Начните с применения `ServiceObserver` для этого — так, чтобы декларация модуля *monitor* выглядела следующим образом:

```
module monitor {
    // [... директивы requires опущены ...]
    // мы убрали зависимости от monitor.observer.alpha и beta – ура!
    uses monitor.observer.ServiceObserver;
}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Второй шаг — объявление директив `provides` в модулях поставщика *monitor.observer.alpha* и *monitor.observer.beta*:

```
module monitor.observer.alpha {
    requires monitor.observer;
    // мы избавились от экспорта модуля monitor.observer.alpha – бинго!
    provides monitor.observer.ServiceObserver
        with monitor.observer.alpha.AlphaServiceObserver;
}
```

Однако это не работает — компилятор выдает ошибку:

```
> The service implementation does not have
> a public default constructor:
>     AlphaServiceObserver
```

Конструкторы поставщика и его методы должны быть без параметров, но `AlphaServiceObserver` ожидает URL сервиса, который должен наблюдать. Как поступить? Можно было бы установить URL после создания, но это сделало бы класс изменчивым и подняло бы вопрос о том, что делать, если сервис не *alpha*. Нет, проще создать фабрику для наблюдателей, которая возвращает экземпляр, только если URL правильный, и делает данную фабрику сервисом.

Итак, создайте новый интерфейс `ServiceObserverFactory` в *monitor.observer*. У него есть единственный метод `createIfMatchingService`, который ожидает URL сервиса и возвращает `Optional<ServiceObserver>`. В *monitor.observer.alpha* и *monitor.observer.beta* создайте реализации, которые делают то же, что и статические методы фабрики в `AlphaServiceObserver` и `BetaServiceObserver`. На рис. 10.5 показана соответствующая часть модульного графа.

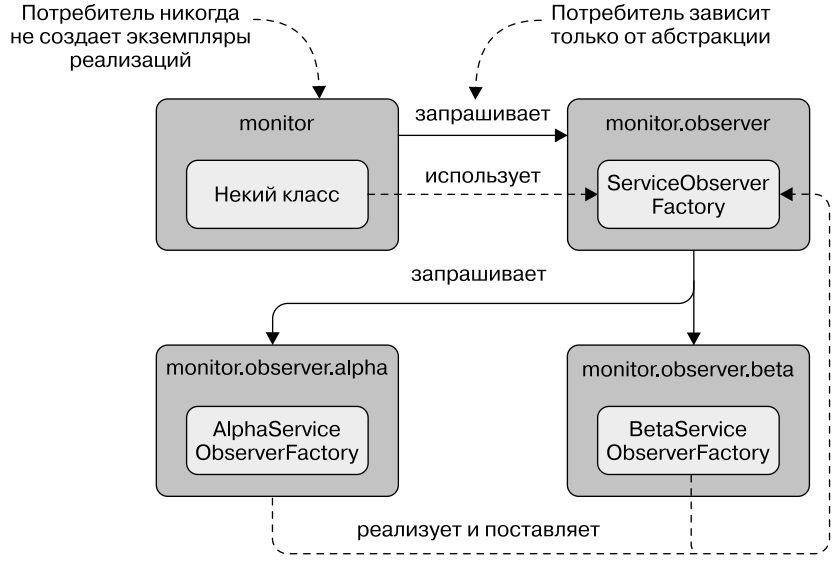


Рис. 10.5. Используя сервисы, monitor зависит только от модуля, определяющего сервис: observer. Модули-поставщики, alpha и beta, больше не требуются напрямую

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

С помощью этих классов можно поставлять и использовать `ServiceObserverFactory` в качестве сервиса. В листинге 10.3 показаны декларации модулей для *monitor*, *monitor.observer*, *monitor.observer.alpha* и *monitor.observer.beta*.

Листинг 10.3. Четыре модуля, работающие с ServiceObserverFactory

```
module monitor {
    requires monitor.observer;
    // [... опущены другие директивы requires ...]
    uses monitor.observer.ServiceObserverFactory;
}

module monitor.observer {
    exports monitor.observer;
}

module monitor.observer.alpha {
    requires monitor.observer;
    provides monitor.observer.ServiceObserverFactory
        with monitor.observer.alpha.AlphaServiceObserverFactory;
}

module monitor.observer.beta {
    requires monitor.observer;
    provides monitor.observer.ServiceObserverFactory
        with monitor.observer.beta.BetaServiceObserverFactory;
}
```

Модуль-потребитель monitor запрашивает monitor.observer, поскольку тот содержит ServiceObserverFactory. Благодаря сервисам не требуются ни alpha, ни beta

Модуль-потребитель monitor использует интерфейс сервиса ServiceObserverFactory

Ничего не изменилось для monitor.observer: он не знает, что используется в качестве сервиса. Все, что нужно, — это обычный экспорт пакета, содержащего ServiceObserver и ServiceObserverFactory

Обоим модулям-поставщикам требуется monitor.observer, поскольку они реализуют содержащиеся в нем интерфейсы — здесь сервисы ничего не изменили

Каждый модуль-поставщик предоставляет сервис ServiceObserverFactory с помощью своего конкретного класса

Последний шаг — получить фабрики *observer* в *monitor*. Для этого вызовите `ServiceLoader.load(ServiceObserverFactory.class)`, соберите данные по возвращенным поставщикам и получите реализации сервисов:

```
List<ServiceObserverFactory> observerFactories = ServiceLoader
    .load(ServiceObserverFactory.class).stream()
    .map(Provider::get)
    .collect(toList());
```

Provider::get создает экземпляр поставщика (см. подраздел 10.4.2)

Вот и все: мы получили кучу поставщиков сервисов и ни модули-потребители, ни модули-поставщики не знают друг о друге. Их единственная связь заключается в том, что все они зависят от модуля API.

Платформенные модули также объявляют и применяют множество сервисов. Особенно интересен `java.sql.Driver`, объявленный и используемый *java.sql*:

```
$ java --describe-module java.sql

> java.sql
# директивы exports опущены
# директивы requires опущены
> uses java.sql.Driver
```

Таким образом, *java.sql* может получить доступ ко всем реализациям `Driver`, предоставленным другими модулями.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Еще один пример применения сервисов в платформе — `java.lang.System.LoggerFinder`. Это часть нового API, добавленного в Java 9, и она позволяет пользователям направлять сообщения журнала JDK (не JVM!) в выбранную ими среду ведения журнала (скажем, Log4J или Logback). Вместо записи в стандартный формат JDK задействует `LoggerFinder` для создания экземпляров `Logger`, а затем записывает в них все сообщения.

Для Java 9 и более поздних версий фреймворки ведения журналов могут реализовывать фабрики для регистраторов, которые используют инфраструктуру фреймворка:

```
public class ForesterFinder extends LoggerFinder {

    @Override
    public Logger getLogger(String name, Module module) {
        return new Forester(name, module);
    }
}
```

Взят из воображаемого фреймворка Forester

Но как фреймворки для ведения журналов могут сообщить *java.base* о своих реализациях `LoggerFinder`? Легко: они предоставляют сервис `LoggerFinder` с собственной реализацией:

```
module org.forester {
    provides java.lang.System.LoggerFinder
        with org.forester.ForesterFinder;
}
```

Это работает, поскольку базовый модуль использует `LoggerFinder`, а затем вызывает `ServiceLoader`, чтобы найти реализации `LoggerFinder`. Он получает специфичный для фреймворка `finder`, просит его создать реализации `Logger`, а затем применяет их для регистрации сообщений.

Все описанное должно дать хорошее представление о том, как создавать и использовать сервисы. Перейдем же к подробностям!

10.2.2. Разрешение модулей для сервисов

Если вы когда-либо запускали простое модульное приложение и наблюдали за тем, что делает система модулей (например, с `--show-module-resolution`, как описано в подразделе 5.3.6), то, возможно, были удивлены количеством разрешенных платформенных модулей. В простом приложении, таком как *ServiceMonitor*, единственными платформенными модулями должны быть *java.base* и, может быть, еще один-два, так почему же их так много? Ответ кроется в сервисах.

ВАЖНАЯ ИНФОРМАЦИЯ

Напомню материал подраздела 3.4.3: во время выполнения доступны только те модули, которые попадают в граф при разрешении. Для полной уверенности, что это относится ко всем наблюдаемым поставщикам сервисов, процесс разрешения учитывает директивы uses и provides. Помимо самого проведения разрешения, описанного в подразделе 3.4.1, после разрешения модуля, потребляющего сервис, он добавляет все обозреваемые модули в граф, который предоставляет данный сервис. Это называется привязкой.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Запуск *ServiceMonitor* с `--show-module-resolution` показывает множество привязок сервисов:

```
$ java
    --show-module-resolution
    --module-path mods:libs
    --module monitor

> root monitor
> monitor requires monitor.observer
# множество разрешений опущено
> monitor binds monitor.observer.beta
> monitor binds monitor.observer.alpha
> java.base binds jdk.charsets jrt:/jdk.charsets
> java.base binds jdk.localedata jrt:/jdk.localedata
# опущено множество других привязок для java.base
# опущен остаток разрешения
```

Модуль *monitor* связывает модули *monitor.observer.alpha* и *monitor.observer.beta*, даже если не зависит ни от одного из них. То же самое происходит с *jdk.charsets*, *jdk.localedata* и многими другими из-за *java.base* и других платформенных модулей. На рис. 10.6 показан модульный граф.

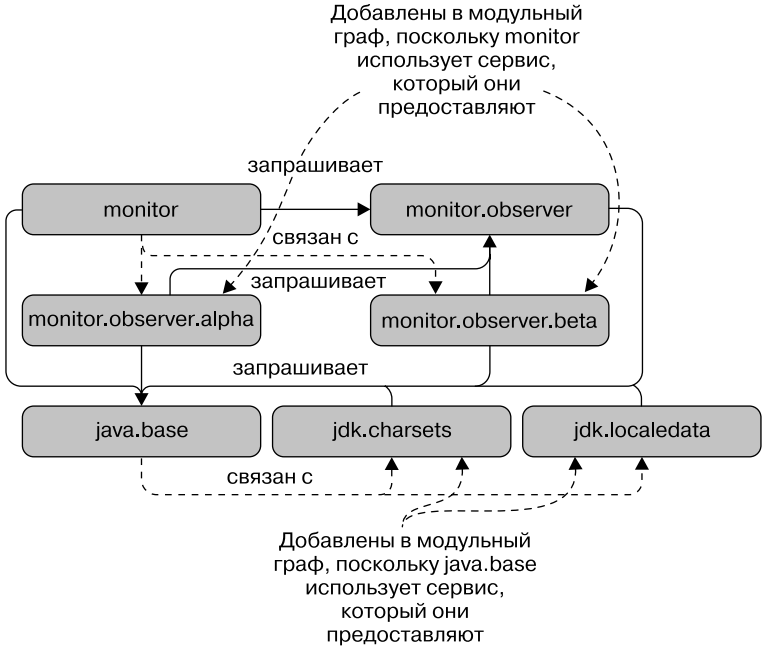


Рис. 10.6. Привязка сервисов — часть разрешения модулей: после разрешения (например, monitor или java.base) анализируются модульные директивы uses, и все модули, которые предоставляют соответствующие сервисы (alpha и beta, как и charsets с localedata) добавляются в модульный граф

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Исключение сервисов с помощью --limit-modules

Сервисы и параметр `--limit-modules` интересно взаимодействуют между собой. Как описано в подразделе 5.3.5, `--limit-modules` ограничивает совокупность обозреваемых модулей до указанных модулей и их переходных зависимостей. Это не касается сервисов! Если модули, предоставляющие сервисы, транзитивно не требуются для модулей, перечисленных после `--limit-modules`, то они не обозреваемы и не попадут в граф. В этом случае вызовы `ServiceLoader::load` часто возвращаются ни с чем.

В случае запуска *ServiceMonitor*, как при проверке разрешения модулей, но с ограничением совокупности обозреваемых модулей, зависимых от *monitor*, вывод будет гораздо более простым:

```
$ java
    --show-module-resolution
    --module-path mods:libs
    --limit-modules monitor
    --module monitor
root monitor
# опущены переходные зависимости модуля monitor
```

Вот и все: ни сервисы, ни фабрики-наблюдатели, ни многие платформенные модули сервисов обычно не связываются. На рис. 10.7 показан этот упрощенный модульный граф.

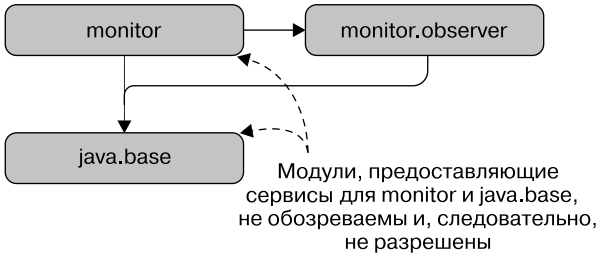


Рис. 10.7. С помощью --limit-modules monitor совокупность обозреваемых модулей ограничилась переходными зависимостями модуля monitor, что исключает поставщики сервисов, разрешенных на рис. 10.6

Особенно эффективно сочетание `--limit-modules` и `--add-modules`: первый может использоваться для исключения всех сервисов, а второй — для добавления желаемых. Это позволяет попробовать различные конфигурации сервисов при запуске, не прибегая к манипуляциям с путем модуля.

Зачем нужны директивы uses

Слегка отклонившись в сторону, я хочу ответить на вопрос, появляющийся у некоторых разработчиков о директиве `uses`: зачем она нужна? Разве система модулей не может самостоятельно искать поставщиков после вызова `ServiceLoader::load`?

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Если модули должным образом отделены с помощью сервисов, то существует большая вероятность, что модули-поставщики не являются транзитивными зависимостями любого корневого модуля. Без дальнейших усилий модули-поставщики сервисов обычно не попали бы в граф и, следовательно, не были бы доступны во время выполнения, когда модуль попытался бы использовать сервис.

Чтобы сервисы работали должным образом, модули-поставщики должны включить их в модульный граф, даже если любой корневой модуль не запрашивает их транзитивно. Но как система модулей может определить, какие модули предоставляют сервисы? Подходят ли под это все модули, имеющие директиву `provides`? Тогда их было бы слишком много. Нет, только поставщики необходимых сервисов должны быть разрешены.

Это делает необходимым определение *uses* в сервисе. Анализ байт-кода, который вызывает `ServiceLoader::load`, медленный и ненадежный, поэтому для гарантии эффективности и правильности требуется более явный механизм: директивы `uses`. Требуя явное объявление того, какие сервисы использует модуль, система модулей может надежно и эффективно сделать все модули-поставщики доступными.

10.3. Эффективное проектирование сервисов

Как вы заметили в разделе 10.2, сервисы — это игра с четырьмя участниками:

- *сервис* — в JPMS — класс или интерфейс;
- *потребитель* — любой фрагмент кода, который хочет использовать сервис;
- *поставщик* — конкретная реализация сервиса;
- *локатор* — «кукловод», который по запросу клиента ищет поставщики и возвращает их. В Java это `ServiceLoader`.

`ServiceLoader` предоставляется JDK (мы более подробно рассмотрим его в разделе 10.4), но при создании сервиса остальные три класса полностью находятся под вашей ответственностью. Какие типы выбрать для сервисов (см. подраздел 10.3.1) и как их лучше спроектировать (см. подраздел 10.3.2)? Не странно ли, что потребители зависят от уродливого глобального состояния (см. подраздел 10.3.3)? Как модули, содержащие сервисы, потребители и поставщики, должны быть связаны друг с другом (см. подраздел 10.3.4)? Чтобы разработать хорошо продуманные сервисы, нужно уметь ответить на эти вопросы.

Мы также рассмотрим использование сервисов для разрыва циклических зависимостей между модулями (см. подраздел 10.3.5). И последнее, но не менее важное — и это особенно интересно для тех, кто планирует задействовать сервисы в разных версиях Java, — мы обсудим, как сервисы работают в простых и модульных JAR-файлах (см. подраздел 10.3.6).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

10.3.1. Типы, которые могут стать сервисами

Сервис может быть конкретным классом (даже конечным), абстрактным классом или интерфейсом. За исключением перечисления, использование конкретного класса (особенно конечного) в качестве сервиса нетрадиционно — весь смысл в том, что модуль должен зависеть от чего-то абстрактного. Сервис должен быть абстрактным классом или интерфейсом, если только другое не требуется в конкретном варианте применения.

> **Об абстрактных классах**
>
> Я не поклонник глубокой иерархии классов и поэтому имею естественное отвращение к абстрактным классам. Благодаря способности Java 8 реализовывать методы в интерфейсах исчез огромный вариант использования абстрактных классов: были предоставлены базовые реализации методов интерфейсов, для которых существует хорошее поведение по умолчанию.
>
> Сейчас я в основном использую их как локальную поддержку (обычно для пакетных или внутренних классов) для реализации сложных интерфейсов, однако не хочу, чтобы они просачивались в публичный API без крайней необходимости. В этом смысле я никогда не создавал сервис, который обязательно является частью открытого API модуля, если это не интерфейс.

10.3.2. Использование фабрик в качестве сервисов

Вернемся к первой попытке рефакторинга архитектуры сервиса наблюдателя для применения сервисов JPMS в подразделе 10.2.1. Она прошла не самым лучшим образом. При использовании интерфейса `ServiceObserver` в качестве сервиса и его реализаций `AlphaServiceObserver` и `BetaServiceObserver` в качестве поставщиков возник ряд проблем.

- Поставщикам нужны методы или конструкторы без параметров, но классы, которые мы хотели использовать, должны были инициализироваться с конкретным состоянием, не предназначенным для модификации.
- Для наблюдателей, которые могут обрабатывать как alpha, так и beta API, было бы неправильно решать, подходят ли они для конкретного сетевого сервиса. Я предпочитаю создавать экземпляры в правильном состоянии.
- Загрузчик сервисов кэширует поставщики (подробнее об этом — в разделе 10.4), поэтому в зависимости от того, как вы используете API, может существовать только единственный экземпляр для каждого поставщика: в данном случае один `AlphaServiceObserver` и один `BetaServiceObserver`.

Это делало непрактичным непосредственное создание нужного нам экземпляра, так что взамен мы использовали фабрику. Оказывается, это не был какой-то особый случай.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Будь то URL-адрес для подключения или имя регистратора — потребитель обычно хочет настроить сервисы, которые использует. Потребитель может также захотеть создать несколько экземпляров какого-либо конкретного поставщика сервисов. В совокупности с требованием загрузчика сервиса к созданию без параметров и свободой кэширования экземпляров это делает нецелесообразным определение используемого типа, `ServiceObserver` или `Logger`, в качестве сервиса.

Вместо этого обычно создают фабрику для желаемого типа, такого как `ServiceObserverFactory` или `LoggerFinder`, и только ее делают сервисом. Согласно фабричному паттерну фабрики несут исключительную ответственность за создание экземпляров в правильном состоянии. Как следствие, их легко спроектировать так, чтобы у них не было собственного состояния, и притом их количество не имеет особого значения. Это делает фабрики подходящими для особенностей `ServiceLoader`.

И у них есть как минимум еще два бонуса:

- если создание экземпляра нужного типа обходится дорого, то наличие фабрики для него в качестве сервиса облегчает потребителям контроль при создании экземпляров;
- если нужно проверить, может ли поставщик обрабатывать определенные входные данные или конфигурацию, фабрику можно снабдить методом, указывающим на это. Как вариант, ее методы могут возвращать тип, указывающий, что создание объекта было невозможно (например, `Optional`).

Я хочу показать два примера выбора сервисов в зависимости от их применимости к определенной ситуации. Первый происходит из *ServiceMonitor*, где в `ServiceObserverFactory` нет метода `create(String)`, возвращающего `ServiceObserver`, но есть метод `createIfMatchingService(String)`, который возвращает `Optional<ServiceObserver>`. Таким образом, можно отправить любой URL любой фабрике, а возвращаемое значение сообщит, можно ли будет его обработать.

В другом примере используется не `ServiceLoader`, а, скорее, аналогичный API в глубине JDK, `ServiceRegistry`. Он был создан исключительно для Java ImageIO API, который использует его, чтобы искать `ImageReader` для переданного изображения в зависимости от его кодека, например JPEG или PNG.

Image IO находит считывателей, запрашивая реализации абстрактного класса `ImageReaderSpi` из реестра, возвращающего экземпляры таких классов, как `JPEGImageReaderSpi` и `PNGImageReaderSpi`. Затем он вызывает `canDecodeInput(Object)` для каждой реализации `ImageReaderSpi`, которая возвращает значение `true`, если изображение использует правильный кодек, как указано в заголовке файла. Только когда реализация вернет `true`, ImageIO вызовет `createReaderInstance(Object)`, чтобы создать фактический считыватель для изображения. На рис. 10.8 показано применение фабрики.

Класс `ImageReaderSpi` действует как фабричный сервис, где `canDecodeInput` используется для выбора правильного поставщика, а `createReaderInstance` — для создания необходимого типа: `ImageReader`. Как показано в подразделе 10.4.2, существует альтернативный подход к выбору подходящего поставщика.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

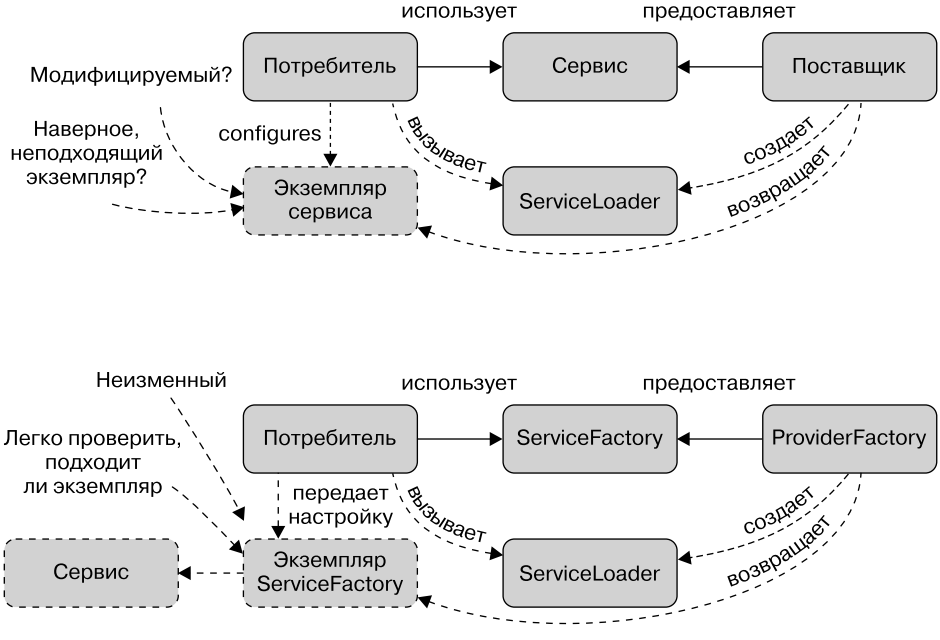


Рис. 10.8. Создание желаемого типа сервиса часто не соответствует особенностям JDK. Вместо этого рассмотрите проектирование фабрики, которая создает экземпляры в правильной конфигурации и делает их сервисами

Таким образом, стоит вначале рассматривать выбор не того типа, который нужно использовать в качестве сервиса, а того (фабрики), который возвращает экземпляры нужного типа. Данная фабрика не должна запрашивать собственное состояние для правильной работы. (Описанное также значительно упрощает реализацию в поточно-ориентированном режиме, если это актуально в вашем случае.) Рассматривайте фабрики как способ отделить исходные требования для типа, который хотите использовать, от конкретных требований инфраструктуры сервисов вместо смешивания их в один тип.

10.3.3. Изоляция потребителей от глобального состояния

Код, вызывающий `ServiceLoader::load`, по сути, сложно тестировать, поскольку он зависит от глобального состояния приложения: модулей, с которыми была запущена программа. Это может легко стать проблемой, когда модуль, использующий сервис, не зависит от модуля, предоставляющего его (как и должно быть), ведь тогда инструмент сборки не будет включать модуль-поставщик в путь тестируемого модуля.

Ручная подготовка `ServiceLoader` к тестированию, чтобы он возвращал определенный список поставщиков сервисов, требует некой доли тяжелой работы.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Это проклятие для тестов, которые должны выполняться изолированно и на небольших блоках кода.

Кроме того, вызов `ServiceLoader::load` обычно не решает проблему, которая волнует пользователя приложения. Это просто необходимый и технический шаг к такому решению. Подобный вариант ставит его на другой уровень абстракции, отличный от кода, который применяет полученные поставщики сервисов. Фанаты принципа единой ответственности сказали бы: такой код имеет две обязанности (запрашивать поставщиков и выполнять бизнес-требования), что кажется излишним.

Эти свойства предполагают, что загрузку сервиса обработки кода не следует смешивать с кодом, реализующим бизнес-требования приложения. К счастью, хранить их раздельно не так уж сложно. Где-то создается экземпляр, который в конечном итоге использует поставщики, и это, как правило, подходящее место для вызова `ServiceLoader` и последующей передачи поставщиков. *ServiceMonitor* имеет ту же структуру: создает все экземпляры, необходимые для запуска приложения в классе `Main` (включая реализации `loadingServiceObserver`), а затем переходит к `Monitor`, который выполняет реальную работу сервисов мониторинга.

Листинги 10.4 и 10.5 отражают сравнение. В листинге 10.4 `IntegerStore` выполняет всю тяжелую работу, которая смешивает ответственности. Это также затрудняет тестирование кода с помощью `IntegerStore`, поскольку тесты должны знать о вызове `ServiceLoader`, а затем убеждаться, что он возвращает нужные экземпляры `IntegerMaker`.

В листинге 10.5 `IntegerStore` подвергается рефакторингу и теперь ожидает, что код, который его создает, доставит `List<IntegerMaker>`. Это делает его код ориентированным на имеющуюся бизнес-проблему (создание целых чисел) и устраняет любую зависимость от `ServiceLoader` и, следовательно, глобального состояния приложения. Протестировать это проще простого. Кто-то все еще должен иметь дело с загрузкой сервисов, но метод `create...`, который вызывается во время установки приложения, — гораздо более подходящее место для данной задачи.

Листинг 10.4. Трудно протестировать из-за слишком большого количества ответственностей

```
public class Integers {

    public static void main(String[] args) {
        IntegerStore store = new IntegerStore();
        List<Integer> ints = store.makeIntegers(args[0]);
        System.out.println(ints);
    }
}

public class IntegerStore {

    public List<Integer> makeIntegers(String config) {
        return ServiceLoader
            .load(IntegerMaker.class).stream()
            .map(Provider::get)
            .map(maker -> maker.make(config))
            .distinct()
```

Результаты этого вызова напрямую зависят от содержимого пути модуля, что затрудняет тестирование

Решает техническое требование загрузки производителей целых чисел

Решает бизнес-задачу: создает уникальные целые числа и сортирует их

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
            .sorted()
            .collect(toList());
    }

}

public interface IntegerMaker {

    int make(String config);

}
```

Листинг 10.5. Переписан для улучшения дизайна и тестируемости

```
public class Integers {

    public static void main(String[] args) {
        IntegerStore store = createIntegerStore();
        List<Integer> ints = store.makeIntegers(args[0]);
        System.out.println(ints);
    }

    private static IntegerStore createIntegerStore() {
        List<IntegerMaker> makers = ServiceLoader
            .load(IntegerMaker.class).stream()
            .map(Provider::get)
            .collect(toList());
        return new IntegerStore(makers);
    }

}

public class IntegerStore {

    private final List<IntegerMaker> makers;

    public IntegerStore(List<IntegerMaker> makers) {
        this.makers = makers;
    }

    public List<Integer> makeIntegers(String config) {
        return makers.stream()
            .map(maker -> maker.make(config))
            .distinct()
            .sorted()
            .collect(toList());
    }

}

public interface IntegerMaker {

    int make(String config);

}
```

Решает техническое требование для загрузки производителей целых чисел во время установки

IntegerStore получает производителей во время создания и не зависит от ServiceLoader

Метод makeIntegers может сфокусироваться на бизнес-требованиях

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В зависимости от конкретного проекта и требований вам может понадобиться передать поставщикам несколько вызовов метода или конструктора, обернуть его в другой объект, который откладывает загрузку до последнего момента, или настроить фреймворк внедрения зависимостей, но все это должно быть выполнимо. И оно того стоит — ваши тесты и коллеги будут вам благодарны.

10.3.4. Организация сервисов, потребителей и поставщиков в модули

После определения типа, дизайна и потребления сервиса возникает вопрос: как можно организовать сервис и двух других участников (потребители и поставщики), в модули? Очевидно, что сервисы должны быть реализованы, и в целях обеспечения ценности коду в модулях, отличных от того, который содержит сервис, следует иметь возможность его реализовать. Это значит, что тип сервиса должен быть общедоступным и находиться в экспортированном пакете.

Потребитель не обязательно должен быть публичным или экспортируемым и, следовательно, может быть внутренним для своего модуля. Однако он должен иметь доступ к типу сервиса, поэтому ему требуется модуль, содержащий сервис (*сервис*, а не классы, его реализующие). Потребитель и сервис нередко оказываются в одном модуле, как в случае с *java.sql* и `Driver`, а также *java.base* и `LoggerFinder`.

Наконец, мы подошли к поставщикам. Поскольку они реализуют сервис, им приходится читать модуль, определяющий его, — это очевидно. Интересный вопрос заключается в том, должен ли предоставляющий тип стать частью публичного API модуля, помимо того, что указано в директиве `provides`.

Поставщик сервиса должен быть публичным, однако нет никаких технических требований для экспорта его пакета — загрузчик сервиса хорош для создания экземпляров недоступных классов. Таким образом, экспорт пакета, содержащего поставщик, без нужды увеличивает поверхность API модуля. Он также предлагает потребителям делать то, что они не должны, например приводить сервис к его реальному типу для доступа к дополнительным функциям (аналогично тому, что произошло с `URLClassLoader`; см. подраздел 6.2.1). Поэтому я советую не делать поставщики сервисов доступными.

Итого (рис. 10.9):

- сервисы должны быть публичными и находиться в экспортированном пакете;
- потребители могут быть внутренними. Они должны читать модуль, определяющий сервис, или даже могут быть частью этого модуля;
- поставщики должны быть публичными, но им не следует быть в экспортированном пакете, чтобы минимизировать злоупотребления и поверхность API. Им нужно читать модуль, определяющий сервис.

ПРИМЕЧАНИЕ

Если вам интересно, то модуль может предоставлять сервис только с тем типом, который ему принадлежит. Реализация сервиса, указанная в директиве provides, должна находиться в том же модуле, что и декларация.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

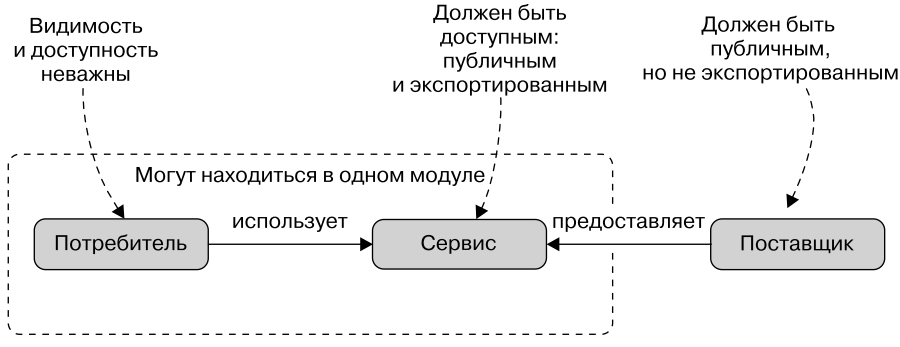


Рис. 10.9. Требования к видимости и доступности для потребителей, сервисов и поставщиков

10.3.5. Использование сервисов для разбиения циклических зависимостей

При работе с кодовой базой, разбитой на подпроекты, всегда наступает момент, когда один из них становится слишком большим и нужно разделить его на более мелкие проекты. Для этого требуется некоторый объем работы, но, имея достаточно времени, чтобы распутать классы, мы чаще всего можем достичь цели. Иногда, однако, код слипается настолько плотно, что невозможно найти способ разделить его на части.

Распространенная причина тому — циклические зависимости между классами. Это могут быть два класса, импортирующие друг друга, или более длинный цикл, включающий несколько классов, каждый из которых импортирует следующий. Однако такой цикл может получиться, только если одни составляющие его классы должны находиться в одном проекте, а другие — в другом. Такая проблема может появиться даже без системы модулей, поскольку инструменты сборки обычно тоже не любят циклические зависимости; но JPMS выражает собственное сильное несогласие.

ПРИМЕЧАНИЕ

Из-за правил доступности, зависимости между классами, которые находятся в отдельных модулях, требуют зависимостей между этими модулями (см. раздел 3.3). Если зависимости класса циклические, то же самое относится и к модулям, и правила читаемости этого не допускают (см. раздел 3.2).

Как поступить? Поскольку вы читаете главу о сервисах, вас может не удивить, что собственно они же и могут помочь. Идея состоит в том, чтобы инвертировать одну из зависимостей в цикле, создав сервис, который находится в зависимом модуле. Вот как это сделать, шаг за шагом (рис. 10.10).

1. Посмотрите на цикл зависимостей модуля и определите, какую из них вы хотите инвертировать. Я назову два задействованных модуля *зависимым* (тот, в котором есть директива `requires`) и *предоставляющим зависимость*. В идеале *зависимый* использует один тип из *предоставляющего зависимость*. Я остановлюсь на этом особом случае — если еще есть типы, то повторите следующие шаги для каждого из них.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

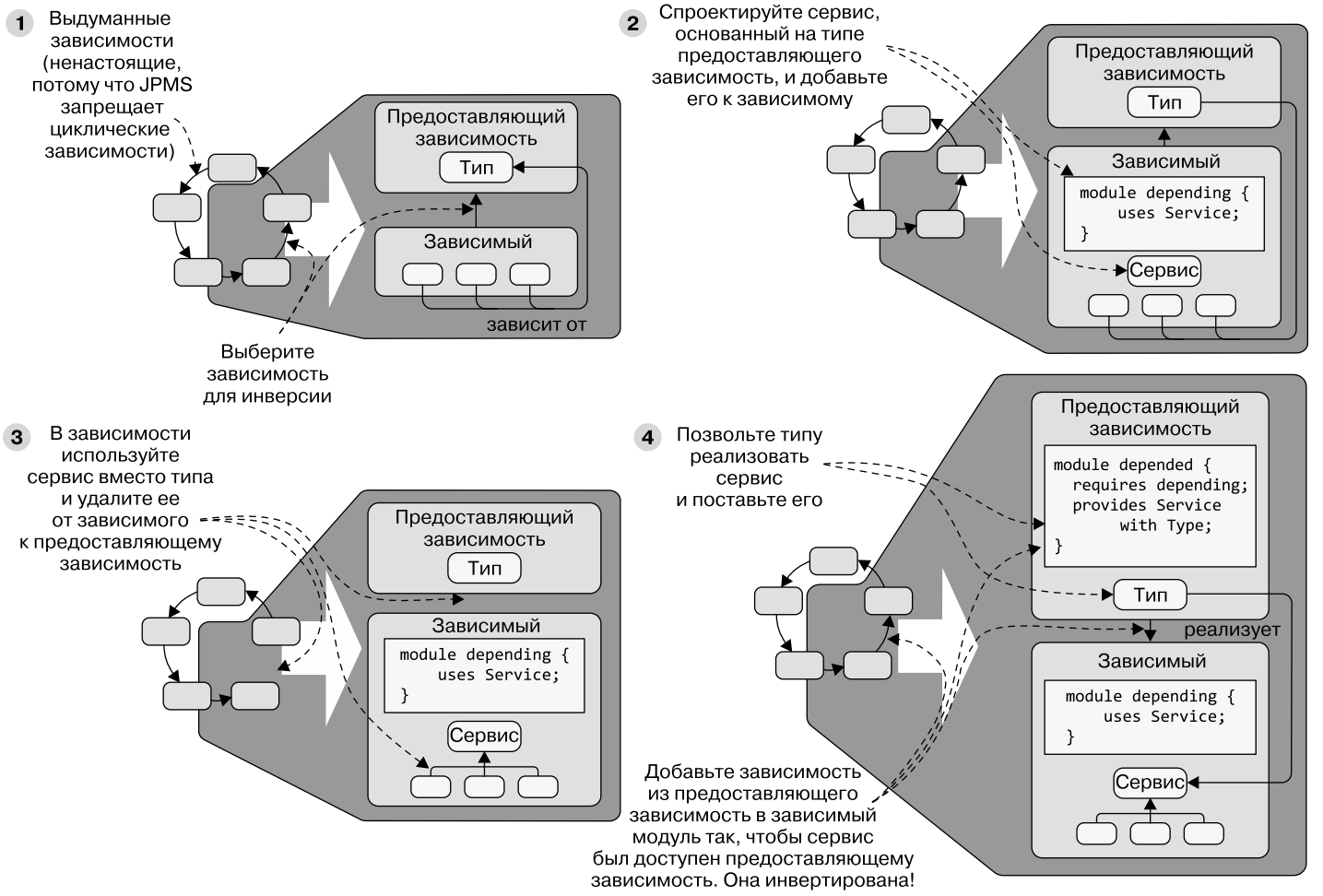


Рис. 10.10. Использование сервисов для разбиения циклической зависимости в четыре этапа: 1) выберите зависимость; 2) предоставьте сервис на зависимой стороне; 3) примените данный сервис на зависимой стороне; 4) поставьте сервис на стороне, предоставляющей зависимость

2. В *зависимом* создайте тип сервиса и расширьте декларацию модуля директивой `uses` для этого типа.
3. В *зависимом* удалите зависимость от *предоставляющего зависимость*. Обратите внимание на возникающие в результате ошибки компиляции, поскольку тип *предоставляющего зависимость* больше не доступен. Замените все ссылки на него типом сервиса:
   - обновите импорт и имена классов;
   - вызовы методов не должны затребовать изменений;
   - вызовы конструктора не заработают сразу, поскольку им нужны экземпляры из *предоставляющего зависимость* модуля. Вот где появляется `ServiceLoader`: используйте его для замены конструкций *предоставляющего зависимость* типа, загрузив только что созданный тип сервиса.
4. В *предоставляющем зависимость* добавьте зависимость от *зависимого*, чтобы тип сервиса стал доступным. Предоставьте этому сервису тип, который изначально вызвал проблему.

Сработало! Мы только что инвертировали зависимость между зависимым и *предоставляющим зависимость* (теперь последний зависит от первого) и таким образом разорвали цикл. Ниже представлены еще несколько подробностей, о которых следует помнить.

- Тип в *предоставляющем зависимость*, используемый *зависимым*, может не быть хорошим кандидатом для сервиса. Если так и есть, то рассмотрите возможность создания для него фабрики, описанную в подразделе 10.3.2, или найдите другую зависимость, которую можно заменить.
- Подраздел 10.3.3 исследует проблему разбрасывания вызовов `ServiceLoader` по всему модулю; эта проблема появляется и здесь. Возможно, нужно изменить код в *зависимом* модуле, чтобы минимизировать количество загрузок.
- Тип сервиса не обязательно должен быть в *зависимом*. Как объясняется в подразделе 10.3.4, тип может находиться в любом модуле. Или, скорее, почти в любом — вы же не хотите помещать его в модуль, провоцирующий создание цикла, например в *предоставляющий зависимость*.
- Самое главное — попытайтесь создать сервис, который стоит сам по себе и представляет собой нечто большее, чем просто прерыватель цикла. На практике может быть больше поставщиков и потребителей, чем два описанных задействованных модуля.

10.3.6. Объявление сервисов в разных версиях Java

Сервисы не новы. Они были введены в Java 6, и механизмы, созданные тогда, все еще действуют сегодня. Имеет смысл посмотреть, как они работают без модулей и, в частности, в простых и модульных JAR-файлах.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Объявление сервисов в META-INF/services

До появления системы модулей сервисы работали почти так же, как и сейчас. Единственное отличие состоит в отсутствии каких-либо деклараций модуля, призванных объявить, что JAR задействует или предоставляет сервис. На стороне использования это нормально — весь код может применять все сервисы, которые ему нужны. Однако на стороне поставки JAR должны были объявить свои намерения, и они сделали это в специальном каталоге в JAR.

Чтобы обычный JAR объявил сервис, выполните следующие простые шаги.

1. Поместите файл с полным именем сервиса в качестве имени файла в `META-INF/services`.
2. В файле перечислите все полностью определенные имена классов, которые реализуют сервис.

В качестве примера создадим третий поставщик `ServiceObserverFactory` в новом предполагаемом простом JAR *monitor.observer.zero*. Для этого сначала необходим конкретный класс `ZeroServiceObserverFactory`, который реализует `ServiceObserverFactory` и имеет конструктор без параметров. Это аналог вариантов *alpha* и *beta*, так что не будем обсуждать их подробно.

Простой JAR не имеет дескриптора модуля для объявления сервисов, которые он предоставляет, но с этой целью можно использовать каталог `META-INF/services`: поместите простой текстовый файл `monitor.observer.ServiceObserverFactory` (полное имя *типа сервиса*) в каталог с одной строкой `monitor.observer.zero.ZeroServiceObserverFactory` (полное имя *типа поставщика*). На рис. 10.11 показано, как это выглядит.

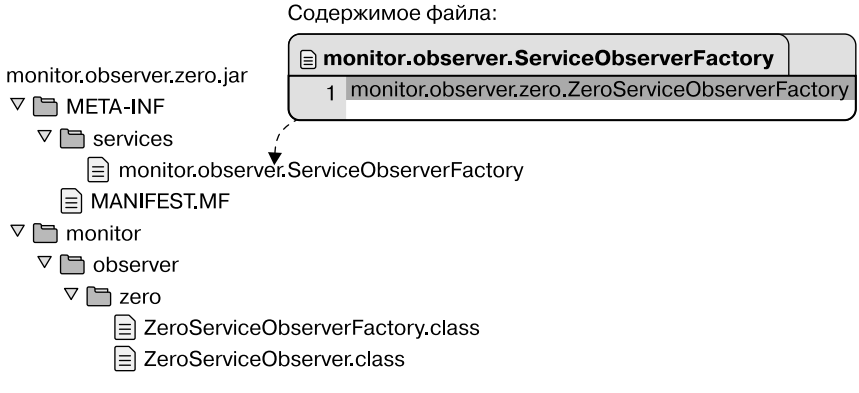


Рис. 10.11. Чтобы объявить поставщики сервисов без модульных деклараций, папка META-INF/services должна содержать простой текстовый файл с названием сервиса и одной строкой для каждого поставщика

Я обещаю, что это работает, и `ZeroServiceObserverFactory` правильно разрешается, когда `Main` передает все фабрики *observer*. Но вам придется поверить мне на слово, пока мы не обсудим, как взаимодействуют простые и модульные сервисы JAR. Об этом еще поговорим.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ПРИМЕЧАНИЕ

Существует небольшое различие между объявлением сервисов в META-INF/services и объявлением их в декларациях модулей. Только последние могут использовать методы поставщика — первые должны придерживаться публичных конструкторов без параметров.

Совместимость между JAR-файлами и путями

Поскольку API загрузчика сервисов существовал до появления системы модулей в Java 9, имеются проблемы совместимости. Могут ли потребители в простых и модульных JAR-файлах использовать сервисы одинаково? И что происходит с поставщиками в разных видах JAR и путей?

Для потребителей сервисов картина проста: явные модули могут использовать сервисы, которые они объявляют директивами `uses`; автоматические модули (см. раздел 8.3) и безымянный модуль (см. раздел 8.2) могут применять все существующие сервисы. Таким образом, это просто работает на стороне потребителя.

Для поставщиков сервисов все немного сложнее. Есть две оси с двумя выражениями в каждой, что приводит к четырем комбинациям:

- тип JAR: простой (объявление сервиса в `META-INF/services`) или модульный (объявление сервиса в дескрипторе модуля);
- тип пути: класса или модуля.

Независимо от того, в каком пути заканчивается простой JAR, загрузчик сервисов будет определять и связывать сервисы в `META-INF/services`. Если JAR находится в пути класса, то его содержимое уже часть безымянного модуля. В случае нахождения его в пути модуля привязка сервиса приводит к созданию автоматического модуля. Это вызывает разрешение всех других автоматических модулей, как описано в подразделе 8.3.2.

Теперь вы знаете, почему можно использовать *monitor.observer.zero*, простой JAR-файл, поставляющий свой сервис в `META-INF/services`, с модульным приложением *ServiceMonitor*. И неважно, какой путь я выберу; это работает для обоих, не вызывая дальнейших хлопот.

ВАЖНАЯ ИНФОРМАЦИЯ

Модульные JAR-файлы в пути модуля — лучшее место для сервисов в системе модулей, поэтому они работают без ограничений. Однако в пути к классам модульные JAR могут вызвать проблемы. Они обрабатываются как простые JAR-файлы, поэтому им нужны сущности в папке META-INF/services. Как разработчик, проект которого опирается на сервисы и чьи модульные артефакты должны работать в обоих путях, вы должны объявить сервисы в дескрипторе модуля и в META-INF/services.

Запуск *ServiceMonitor* из пути класса не приводит к полезному выводу, поскольку фабрика наблюдателя не может быть найдена, если только вы не добавите

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

monitor.observer.zero в общий список. Благодаря определению поставщика в `META-INF/services` он хорошо подходит для работы с безымянным модулем, и это действительно так — в отличие от поставщиков *alpha* и *beta*.

10.4. Доступ к сервисам с помощью ServiceLoader API

Несмотря на существование со времен Java 6, `ServiceLoader` не получил широкого распространения, но, учитывая его выдающуюся интеграцию с системой модулей, я ожидаю, что его использование значительно возрастет. Для полной уверенности в том, что вы будете знать данный API, мы рассмотрим его в этом разделе.

Как обычно, первый шаг — ознакомиться с основами, что в данном случае не займет много времени. Однако у загрузчика сервисов есть особенности, и для подтверждения того, что они вас не смутят, мы обсудим и их.

10.4.1. Загрузка и доступ к сервисам

Использование `ServiceLoader` — всегда двухэтапный процесс.

1. Создайте экземпляр `ServiceLoader` для необходимого сервиса.
2. Задействуйте этот экземпляр для доступа к поставщикам сервиса.

Кратко рассмотрим каждый шаг, чтобы вы знали все варианты. Проверьте также табл. 10.1, в которой представлен обзор всех методов `ServiceLoader`.

Таблица 10.1. Краткий обзор API ServiceLoader API

Возвращаемый тип	Имя метода	Описание
Методы для создания нового загрузчика сервиса для заданного типа		
ServiceLoader<S>	load(Class<S>)	Загружает поставщики, начиная с загрузчика класса контекста текущего потока
ServiceLoader<S>	load(Class<S>, ClassLoader)	Загружает поставщики, начиная с загрузчика указанного класса
ServiceLoader<S>	load(ModuleLayer, Class<S>)	Загружает поставщики, начиная с модулей в данном модульном слое
ServiceLoader<S>	loadInstalled(Class<S>)	Загружает поставщики из загрузчика платформенного класса
Методы доступа к поставщикам сервисов		
Optional<S>	findFirst()	Загружает первый доступный поставщик
Iterator<S>	iterator()	Возвращает итератор для ленивой загрузки и создания экземпляров доступных поставщиков
Stream<Provider<S>>	stream()	Возвращает поток для ленивой загрузки доступных поставщиков
Void	reload()	Очищает кэш поставщиков этого загрузчика, чтобы все поставщики были перезагружены

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Способы создания ServiceLoader

Первый шаг, создание экземпляра `ServiceLoader`, покрывается несколькими статическими методами `load`. Самому простому требуется лишь экземпляр `Class<S>` для сервиса, который нужно загрузить (это называется *токеном типа*, в данном случае для типа `S`):

```
ServiceLoader<TheService> loader = ServiceLoader.load(TheService.class);
```

Другие методы `load` нужны, только если вы манипулируете несколькими загрузчиками классов или слоями модулей (см. раздел 12.4); это нечастый случай, так что я не буду вдаваться в подробности. Документации API для соответствующих перегрузок в полной мере освещают тему.

Еще один метод получает загрузчик сервиса: `loadInstalled`. Здесь это интересно, поскольку он ведет себя специфически: игнорирует пути модуля и класса и загружает лишь сервисы из платформенных модулей, то есть возвращаются только поставщики, найденные в модулях JDK.

Доступ к поставщикам сервисов

При наличии экземпляра `ServiceLoader` для нужного сервиса пришло время начать использовать его поставщиков. Это можно сделать с помощью двух с половиной методов:

- `Iterator<S> iterator()` позволяет выполнять итерации по созданным поставщикам сервисов;
- `Optional<S> findFirst()` применяет `iterator` для *возврата первого поставщика*, если таковой был найден (удобный метод, поэтому я считаю его только половиной);
- `Stream<Provider<S>> stream()` позволяет выполнять *потоковую передачу по поставщикам сервисов*, которые включены в экземпляр `Provider`. (Что с ним не так? Объясню в подразделе 10.4.2.)

При наличии особой потребности в ленивой загрузке/кэшировании (подробнее см. в подразделе 10.4.2) можно сохранить экземпляр `ServiceLoader`. Но в большинстве случаев это не нужно и можно сразу же начать перебирать или передавать поставщики:

```
ServiceLoader
    .load(TheService.class)
    .iterator()
    .forEachRemaining(TheService::doTheServiceThing);
```

Если вас интересует несоответствие между `iterator`, перечисляющим `S` и `stream`, перечисляющим `Provider<S>`, — на то есть исторические причины: хоть `iterator` и существует со времен Java 6, `stream` и `Provider` были добавлены только в Java 9.

Есть одна деталь, очевидная, стоит задуматься о ней, но все же легко упускаемая из виду — для данного сервиса может не быть поставщика. `Iterator` и `stream` могут быть пустыми, а `findFirst` может возвращать пустой `Optional`. Если вы фильтруете

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

список по возможностям, как описано в подразделах 10.3.2 и 10.4.2, то результат с нулевым числом подходящих поставщиков будет еще более вероятным.

Убедитесь, что ваш код либо изящно обрабатывает данный случай и может функционировать при отсутствии сервиса, либо быстро останавливает работу. Раздражает ситуация, когда приложение игнорирует легко обнаруживаемую ошибку и продолжает работать в нежелательном и неожиданном состоянии.

10.4.2. Особенности загрузки сервисов

API `ServiceLoader` довольно прост, однако не обманывайте себя. За «кулисами» происходит несколько важных вещей, и нужно знать об этом в момент использования API для чего-либо, кроме базовых примеров Hello, services!. Это касается ленивой загрузки сервисов, возможностей их параллелизма (или отсутствия таковой) и правильной обработки ошибок. Пройдемся по каждому пункту.

Лень и выбор подходящего поставщика

Загрузчик сервиса настолько ленив, насколько это возможно. Вызванный на `ServiceLoader<S>` (где `S` — это тип сервиса, с которым был вызван `ServiceLoader::load`), его метод `iterator` возвращает `Iterator<S>`, который находит и создает следующий поставщик лишь при вызове `hasNext` или `next`.

Метод `stream` еще более ленив. Он возвращает `Stream<Provider<S>>`, который не только лениво находит поставщиков (например, `iterator`), но также возвращает экземпляры `Provider`, дополнительно откладывающие создание экземпляров сервиса до тех пор, пока не будет вызван их метод `get`. Их метод `type` дает доступ к экземпляру `Class<? extends S>` для своего конкретного поставщика (имеется в виду тип, *реализующий* сервис, а *не являющийся* им).

Доступ к типу поставщика полезен для сканирования аннотаций без фактического экземпляра класса. Подобно тому, что мы обсуждали в конце подраздела 10.3.2, это дает инструмент выбора подходящего поставщика сервиса для данной конфигурации, но без возможного влияния на производительность при его создании в первую очередь. Вот почему аннотированность класса дает некоторое представление о том, подходит ли данный поставщик.

В продолжение примера использования *ServiceMonitor* для фабрик `ServiceObserver`, применимых к конкретным поколениям REST-сервисов, фабрики можно аннотировать с помощью `@Alpha` или `@Beta`, чтобы указать поколение, для которого они были созданы:

```
Optional<ServiceObserverFactory> alphaFactory = ServiceLoader
    .load(ServiceObserverFactory.class).stream()
    .filter(provider -> provider.type().isAnnotationPresent(Alpha.class))
    .map(Provider::get)
    .findFirst();
```

Здесь `Provider::type` используется для доступа к `Class<? extends ServiceObserver>`, о котором затем с помощью `isAnnotationPresent` вы узнаете, был ли

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

он аннотирован с `@Alpha`. Только когда вызывается `Provider::get`, создается фабрика.

На пике лени экземпляр `ServiceLoader` кэширует загруженные до сих пор поставщики и всегда возвращает одни и те же. Однако у него есть метод `reload`, который очищает кэш и запускает новые экземпляры при следующем вызове `iterate`, `stream` или `findFirst`.

Использование одновременных ServiceLoader

Экземпляры `ServiceLoader` не являются потокобезопасными. Допустим, несколько потоков должны работать одновременно с набором поставщиков сервисов либо каждому из них необходимо выполнить один и тот же вызов `ServiceLoader::load`. В таком случае они получают собственный экземпляр `ServiceLoader` — или же вам нужно будет сделать один вызов для всех из них и сохранить результаты в потокобезопасной коллекции.

Обработка ошибок при загрузке сервисов

Все может пойти не так, когда `ServiceLoader` пытается определить местонахождение или создать экземпляр поставщика сервисов:

- поставщик может не выполнить все требования. Возможно, он не реализует тип сервиса или не имеет подходящего метода или конструктора поставщика;
- конструктор или метод поставщика может выдать исключение или (в случае метода) вернуть `null`;
- файл в `META-INF/services` может нарушать требуемый формат или не обрабатываться по другим причинам.

И это лишь очевидные проблемы.

Поскольку загрузка выполняется лениво, `load` не может генерировать никакие исключения. Вместо этого методы итератора `hasNext` и `next`, а также потоковая обработка и методы `Provider` могут генерировать ошибки. Все они будут иметь тип `ServiceConfigurationError`, вследствие чего перехват такой ошибки позволяет решать все потенциальные проблемы.

Резюме

- Архитектура сервиса состоит из четырех частей:
  - *сервис* — класс или интерфейс;
  - *поставщик* — конкретная реализация сервиса;
  - *потребитель* — любой фрагмент кода, который хочет использовать сервис;
  - `ServiceLoader` создает и возвращает экземпляр каждого поставщика данного сервиса потребителям.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- Требования и рекомендации для типа сервиса выглядят следующим образом.
  - Любой класс или интерфейс может быть сервисом, но, поскольку основная цель — обеспечить максимальную гибкость для потребителей и поставщиков, рекомендуется использовать интерфейсы (или по крайней мере абстрактные классы).
  - Типы сервисов должны быть публичными и находиться в экспортированном пакете. Таким образом, они становятся частью публичного API их модуля и их следует разрабатывать и сопровождать соответствующим образом.
  - Декларация модуля, определяющего сервис, не содержит записи, чтобы пометить тип как сервис. Тип становится сервисом из-за потребителей и поставщиков, использующих данный тип как единое целое.
  - Сервисы редко появляются случайно, они всегда создаются для конкретных случаев применения. Всегда задумывайтесь о том, чтобы сделать используемый тип не сервисом, а фабрикой для него. Это облегчает поиск подходящей реализации, а также позволяет контролировать момент создания экземпляров и их состояние.
- Требования и рекомендации для поставщиков изложены ниже.
  - Модули, поставляющие сервисы, должны иметь доступ к типу сервиса, поэтому должны запрашивать содержащий его модуль.
  - Существует два способа создания поставщика сервиса: конкретный класс, который реализует тип сервиса и имеет конструктор поставщика (публичный, конструктор без параметров), или тип с методом поставщика (публичный, статический метод без параметров, называемый `provide`), который возвращает экземпляр, реализующий тип сервиса. В любом случае тип должен быть публичным, но нет нужды экспортировать содержащий его пакет. Наоборот, желательно не включать тип поставщика в публичный API модуля.
  - Модули, предоставляющие сервисы, объявляют об этом, добавляя директиву `${`*`сервис`*`} with ${`*`поставщик`*`}`к своему дескриптору.
  - Если модульный JAR должен поставлять сервисы, даже будучи размещенным в пути к классам, ему также нужны записи в каталоге `META-INF/services`. Для каждого из них предоставляется директива `${`*`сервис`*`} with ${`*`поставщик`*`}`, создается простой файл с именем `${`*`сервис`*`}`, который содержит одну строку с `${`*`поставщик`*`}` (все имена должны быть полностью квалифицированы).
- Требования и рекомендации для потребителей таковы.
  - Модулям, применяющим сервисы, надлежит иметь доступ к типу сервиса, поэтому они должны запрашивать содержащий его модуль. Им не следует запрашивать модули, предоставляющие данный сервис, — напротив, это будет противоречить основной цели использования сервисов: разъединению потребителей и поставщиков.
  - Нет ничего плохого в том, что типы и потребители сервисов находятся в одном модуле.
  - Любой код может применить сервисы независимо от собственной доступности, но содержащий его модуль должен объявить, какие сервисы использует,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

с помощью директивы `uses`. Это позволяет системе модулей эффективно выполнять привязку сервисов и делает модульные декларации более явными и читабельными.

- Модули потребляются путем вызова `ServiceLoader::load`, а затем итерации или потоковой передачи по возвращенным экземплярам путем вызова `iterate` или `stream`. Вполне возможно, что поставщики не найдутся, и потребители должны изящно разобраться с этим.
- Поведение кода, потребляющего сервисы, зависит от глобального состояния: какие модули поставщиков присутствуют в модульном графе. Это дает подобному коду нежелательные свойства, например, усложняет тестирование. Попробуйте вставить загрузку сервиса в установочный код, который создает объекты в их правильной конфигурации (например, в фреймворке внедрения зависимостей), и всегда разрешайте коду поставщика передавать поставщики сервисов в потребляющие классы (скажем, во время построения).
- Загрузчик сервиса создает поставщики как можно позже. Его метод `stream` даже возвращает `Stream<Provider<S>>`, где `Provider::type` может использоваться для доступа к экземпляру `Class` поставщика. Это позволяет искать подходящий поставщик, проверяя аннотации на уровне класса без создания его экземпляра.
- Экземпляры загрузчика сервисов не являются потокобезопасными. Если вы используете их одновременно, то обеспечьте синхронизацию.
- Все проблемы при загрузке и создании экземпляров поставщиков посылаются как `ServiceConfigurationError`. Из-за лени загрузчика это происходит не во время загрузки, но позже в `iterate` или `stream`, когда встретятся проблемные поставщики. Обязательно помещайте все взаимодействие с `ServiceLoader` в блок `try` для обработки ошибок.

- И наконец, пару пунктов о разрешении модуля и т. п.
  - Когда процесс разрешения модулей обрабатывает модуль, который объявляет об использовании сервиса, все модули, предоставляющие этот сервис, разрешаются и, таким образом, включаются в модульный граф приложения. Это называется *привязкой сервисов* и вместе с применением сервисов в JDK объясняет, почему по умолчанию даже небольшие приложения задействуют множество платформенных модулей.
  - Параметр командной строки `--limit-modules`, с другой стороны, не выполняет привязку сервисов. Как следствие, поставщики, не являющиеся переходными зависимостями модулей, предоставленных этому параметру, не попадают в граф и недоступны во время выполнения. Параметр можно использовать для исключения сервисов, не обязательно вместе с `--add-modules`, чтобы добавить некоторые из них обратно.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

11 Уточнение API и зависимостей

В этой главе

- Обработка зависимостей, которые являются частью API модуля.
- Агрегирование и рефакторинг модулей без прерывания клиентов.
- Определение необязательных зависимостей.
- Написание кода с отсутствующими зависимостями.
- Экспорт пакетов только в выбранные модули.

Глава 3 объясняет, почему директивы `requires` и `exports` являются основой читабельности и доступности. Но эти механизмы строги: *каждый* модуль должен запрашиваться явным образом, *все* требуемые модули должны присутствовать, чтобы скомпилировать и запустить приложение, а экспортированные пакеты доступны для *всех* остальных модулей. Этого достаточно для большинства случаев использования, но все же есть значительная часть ситуаций, где эти решения слишком огульны.

Наиболее очевидный вариант использования — это необязательные зависимости, необходимые модулю для компиляции, но не обязательно присутствующие во время выполнения. Spring, например, делает это с помощью библиотеки связей данных Jackson. При запуске приложения Spring, если вы хотите применить JSON в качестве формата передачи данных, можно получить его поддержку, задействовав артефакт Jackson. С другой стороны, при отсутствии данного артефакта Spring продолжает работать — но без поддержки JSON. Spring *использует* Jackson, однако не *требует* его.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Обычные директивы `requires` не охватывают этот вариант использования, поскольку модули должны присутствовать для запуска приложения. В некоторых случаях это можно решить с помощью сервисов, но применение их для всех необязательных зависимостей может привести к множеству неудобных и сложных реализаций. Следовательно, простое выражение, что зависимость не требуется во время выполнения, является важной особенностью; раздел 11.2 показывает, как JPMS реализует это.

Другой случай использования, когда строгость системы модулей может стать помехой, — это рефакторинг модулей с течением времени. В любом приличных размеров проекте архитектура развивается постепенно и разработчики захотят объединить или разделить модули. Но что происходит с кодом, зависящим от старых модулей? Не будет ли отсутствовать функциональность (если была разделена между новыми модулями) или даже целые модули (если были объединены)? К счастью, система модулей предлагает функцию, называемую *подразумеваемой читабельностью*, которую можно использовать в таком случае.

Хотя механизмы `requires` и `exports`, уже известные нам, создают сравнительно простую ментальную модель, они не предлагают элегантных решений для случаев использования, не вписывающихся в их подходы «один размер для всех». В этой главе мы рассмотрим такие конкретные случаи и решения, которые предлагает система модулей.

Ко времени их проработки вы сможете задействовать более совершенные механизмы для доступа к зависимостям и функциям экспорта. Это позволит вам, помимо прочего, выражать необязательные зависимости (см. раздел 11.2), делать рефакторинг модулей (см. раздел 11.1) и обмениваться кодом в определенном наборе модулей, сохраняя его закрытым от другого кода (см. раздел 11.3).

11.1. Подразумеваемая читабельность: передача зависимостей

В разделе 3.2 мы подробно изучили, как директивы `requires` устанавливают зависимости между модулями и как их использует система модулей для создания ребер чтения (в конечном итоге получим модульный граф, как показано в подразделах 3.4.1 и 3.4.2). В разделе 3.3 мы увидели, что доступность основана на данных ребрах и для доступа к типу запрашивающий модуль должен прочитать модуль, в котором этот тип содержится (тип также должен быть публичным, а пакет экспортирован, но здесь это не имеет значения).

В текущем разделе мы рассмотрим другой способ предоставления модулям доступа к другим модулям. Мы начнем с обсуждения мотивирующего варианта использования, прежде чем я представлю новый механизм, и разработаем некоторые рекомендации по наилучшему его применению. В конце вы увидите, насколько он эффективен и как может помочь в случаях, гораздо более сложных, чем первоначальные примеры.

Посмотрите ветку `feature-implied-readability` в *ServiceMonitor* — там находится код, сопровождающий этот раздел.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

11.1.1. Выявление зависимостей модуля

Когда речь заходит о взаимодействии между директивами `requires` и доступностью, следует обратить внимание на одну тонкую деталь: *директивы* `requires` создают ребра чтения, но *ребра* являются обязательным условием доступности. Разве это не вызывает вопроса о том, какие другие механизмы могут обеспечить читабельность и, таким образом, разблокировать доступ к типам? Это выходит за рамки теоретических размышлений — подходя к ситуации с практической точки зрения, мы оказываемся на том же месте.

Вернемся к приложению *ServiceMonitor*, в частности к модулям *monitor.observer* и *monitor.observer.alpha*. Предположим, что новый модуль (назовем его *monitor.peek*), хочет напрямую использовать *monitor.observer.alpha*. Ему абсолютно точно не нужен *monitor.observer* или архитектура сервиса, который мы создавали в предыдущей главе. Может ли *monitor.peek* просто запросить *monitor.observer.alpha* и начать его применять?

```
ServiceObserver observer = new AlphaServiceObserver("some://service/url");
DiagnosticDataPoint data = observer.gatherDataFromService();
```

Похоже, ему нужны типы `ServiceObserver` и `DiagnosticDataPoint`. Оба находятся в *monitor.observer*, что же произойдет, если для *monitor.peek* не требуется *monitor.observer*? Он не может получить доступ к его типам, и это приводит к ошибкам компиляции. Как вы видели в процессе обсуждения инкапсуляции переходных зависимостей в подразделе 3.3.2, это особенность системы модулей.

Хотя в данном случае это скорее препятствие. Без типов из *monitor.observer* *monitor.observer.alpha* фактически бесполезен; и каждый модуль, который хочет его использовать, должен также читать *monitor.observer*. (Это показано на рис. 11.1.) Следует ли каждому модулю, применяющему *monitor.observer.alpha*, также указывать `requires` для *monitor.observer*?

Это неудобное решение. Вот если бы имелся еще один механизм для установки читабельности и, таким образом, разблокировки доступа к типам.

Происходящее в примере выше — распространенное явление. Модуль *exposing* зависит от некоего модуля *exposed*, но использует типы из *exposed* в собственном публичном API (как определено в разделе 3.3). В подобных случаях говорят, что *exposing* выставляет свою зависимость от *exposed* своим клиентам, поскольку и они должны зависеть от *exposed*, чтобы применять *exposing*.

Немного упростить данную ситуацию поможет понимание определений на рис. 11.2. Убедитесь в этом. Я буду придерживаться указанных условий при описании задействованных модулей:

- модуль, выставляющий свою зависимость, называется *выставляющим (exposing)* модулем.
- модуль, выставляющийся как зависимость, является *выставленным (exposed)* модулем.
- модуль, зависимый от этого беспорядка, называется *зависимым (depending)* модулем.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

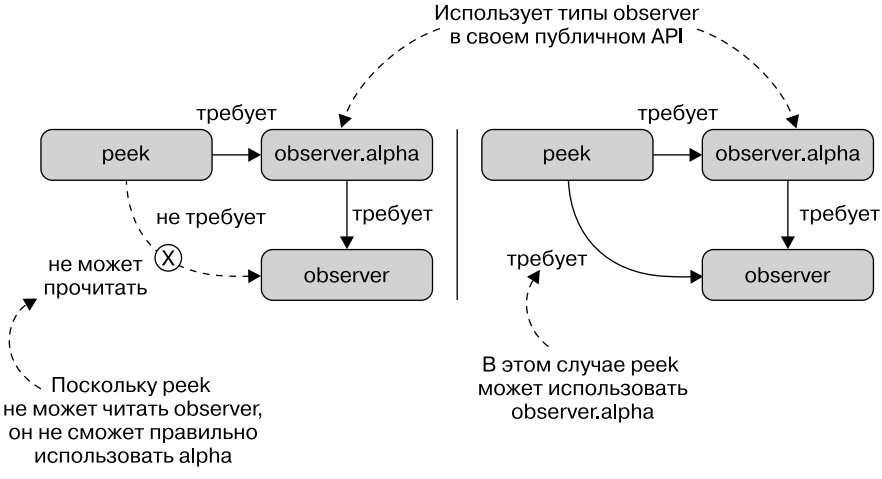


Рис. 11.1. Модуль peek использует observer.alpha, который задействует типы из observer в своем публичном API. Если peek не требует observer *(слева)*, то не может читать его типы, что делает observer.alpha бесполезным. С обычными директивами requires единственным способом обхода становится случай, когда peek также запрашивает observer *(справа)*, что становится громоздким в ситуации применения большого количества модулей

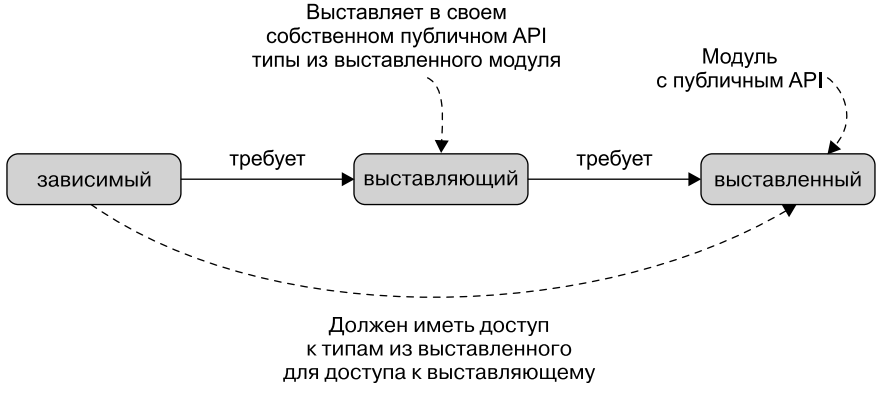


Рис. 11.2. В проблему открытых зависимостей вовлечены три модуля: невиновный, который предоставляет некоторые типы (выставленный; *справа*), виновный, использующий эти типы в своем публичном API (выставляющий; *по центру*), и затронутый, имеющий доступ к типам невиновного (зависимый; *слева*)

Множество примеров можно найти в JDK. Так, модуль *java.sql* содержит тип `java.sql.SQLXML` (используемый, помимо прочего, в `java.sql.Connection`), который задействует типы из модуля *java.xml* в своих публичных методах. Тип `java.sql.SQLXML` — публичный и находится в экспортированном пакете, поэтому является частью API *java.sql*. Это означает следующее: любой зависимый модуль сможет правильно использовать выставляющий модуль *java.sql* только при условии, что прочтет выставленный *java.xml*.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

11.1.2. Модификатор transitive: подразумеваемая читабельность зависимости

Глядя на ситуацию, мы понимаем, что разработчики *выставляющего* модуля — это те, кому нужно решить возникшую проблему. В конце концов, они решают использовать типы *выставленных* модулей в собственном API, заставляя модули, зависимые от них, читать выставленные модули.

Решение для этих ситуаций — использование директивы `requires transitive` в декларации *выставляющего* модуля. Если он объявляет `requires transitive exposed`, то любой модуль, читающий *выставляющий* модуль, будет также неявно читать *выставленный*. Эффект называется подразумеваемой читабельностью: чтение *выставляющего* модуля подразумевает чтение *выставленного*. На рис. 11.3 показана данная директива.

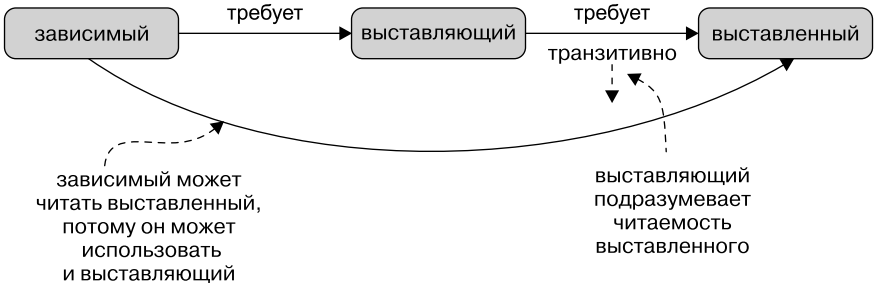


Рис. 11.3. Когда выставляющий модуль использует директиву requires transitive для получения зависимости от выставленного, чтение выставляющего модуля подразумевает чтение выставленного. Как следствие, модули, подобные зависимому *(слева)*, могут считывать данные выставленного модуля, даже если запрашивают только выставляющий

Использование подразумеваемой читабельности очевидно при просмотре декларации модуля или его дескриптора. Изучите *java.sql*, учитывая полученные из подраздела 5.3.1 навыки. Листинг 11.1 показывает, что зависимость от *java.xml* помечена как `transitive`.

Листинг 11.1. Дескриптор модуля java.sql: подразумевает читабельность java.xml и java.logging

```
$ java --describe-module java.sql

> java.sql@9.0.4
> exports java.sql
> exports javax.sql
> exports javax.transaction.xa
> requires java.base mandate
> requires java.logging transitive
> requires java.xml transitive
> uses java.sql.Driver
```

Данные директивы указывают, что модули, читающие java.sql, также могут читать java.xml и java.logging

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Аналогично зависимость от *java.logging* помечается как `transitive`. Причина — открытый интерфейс `java.sql.Driver` и его метод `Logger getParentLogger()`. Он вставляет тип `java.util.logging.Logger` из *java.logging* в публичном API *java.sql*, поэтому *java.sql* подразумевает читабельность *java.logging*. Обратите внимание: хотя java `--describe-module` помещает `transitive` в последнюю очередь, модульная декларация ожидает, что модификатор будет находиться между `requires` и именем модуля (`requires transitive ${`*модуль*`}`).

Возвращаясь к мотивирующему примеру того, как сделать *monitor.observer.alpha* пригодным для использования без зависимости от модулей, также требующих *monitor.observer*, решение теперь очевидно: задействуйте `requires transitive` для объявления зависимости *monitor.observer.alpha* от *monitor.observer*.

```
module monitor.observer.alpha {
    requires transitive monitor.observer;
    exports monitor.observer.alpha;
}
```

Исследуя надежную конфигурацию и отсутствующие зависимости в подразделе 3.2.2, мы обнаружили: хотя во время выполнения требуется, чтобы все зависимости (прямые и косвенные) были обозреваемыми, компилятор предписывает это только для прямых зависимостей. То есть можно скомпилировать *выставляющий* модуль без присутствия его зависимостей. Ну а теперь — как в это вписывается подразумеваемая читабельность?

ВАЖНАЯ ИНФОРМАЦИЯ

Модули, чья читабельность подразумевается для компилируемого модуля, попадают в область «должны быть обозреваемыми». Это говорит о том, что каждая зависимость выставляющего модуля, запрашиваемая транзитивно, как это было с выставленной в предыдущих примерах, должна быть обозреваемой при компиляции выставляющего модуля.

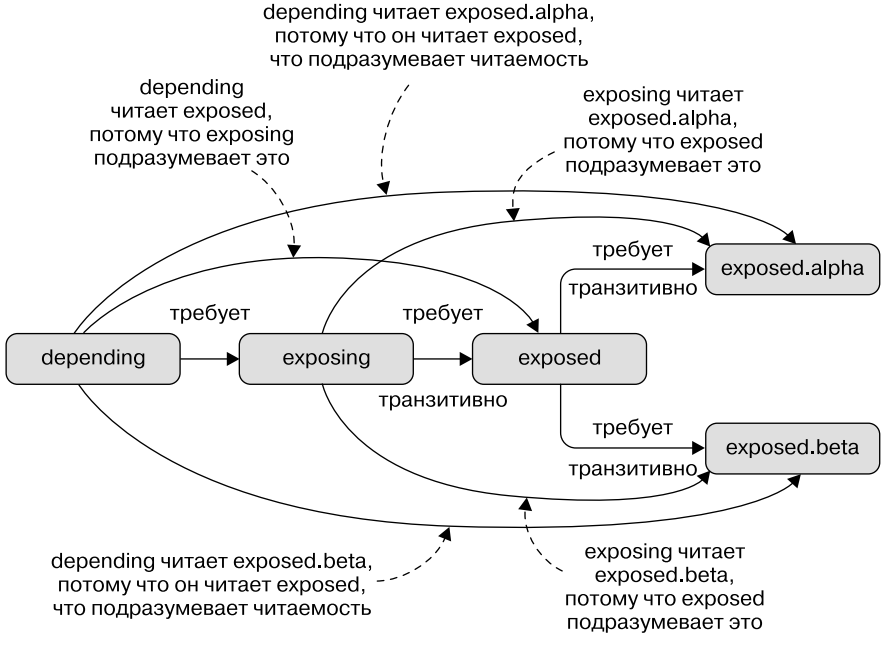
Это происходит независимо от того, используете ли вы типы из *выставленного* модуля, что на первый взгляд может показаться слишком строгим. Но вспомните, в подразделе 3.4.1 сказано, что модули разрешаются, а граф модулей строится до компиляции кода. Модульный граф — основа для компиляции, а не наоборот, и изменение его в зависимости от обнаруженных типов противоречит цели надежной конфигурации. Следовательно, граф модуля должен всегда содержать транзитивные зависимости.

Цепочки зависимостей

Можно задаться вопросом: что произойдет в цепочке зависимостей, где каждая директива `requires` использует `transitive`? Будет ли читабельность подразумеваться в более длинных путях? Ответ — да. Неважно, читается ли выставляющий модуль из-за явной зависимости или подразумеваемой читабельности — это будет в одинаковой

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

мере означать читабельность его зависимостей. Следующий рисунок иллюстрирует транзитивность `transitive`.



Модуль depending запрашивает exposing, что подразумевает читабельность exposed, что, в свою очередь, подразумевает читабельность exposed.alpha и exposed.beta. Подразумеваемая читабельность транзитивна, поэтому модуль depending может прочитать все остальные четыре модуля, хотя зависит только от одного из них

11.1.3. Когда использовать подразумеваемую читабельность

Как вы уже заметили, подразумеваемая читабельность снижает необходимость в явных директивах `requires` в зависимых модулях. Это может быть полезно, но я хочу вернуться к тому, что упомянул ранее лишь мимоходом. Подразумеваемая читабельность идет вразрез с особенностью системы модулей: инкапсуляцией переходных зависимостей, обсуждаемых в подразделе 3.2.2. Имея два противоположных требования (строгость и удобство) и две функции для их выполнения (`requires` против `requires transitive`), важно тщательно рассмотреть компромиссы.

Ситуация похожа на использование модификаторов видимости. Для удобства было бы легко сделать публичным каждый класс, каждое поле и каждый метод. Однако мы этого не делаем, поскольку знаем, что меньшее раскрытие уменьшает поверхность контакта между различными частями кода и облегчает изменение, за-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

мену и повторное использование. И подобно раскрытию типа или члена раскрытие зависимости становится частью публичного API данного модуля и клиенты могут полагаться на подразумеваемую читабельность. Это может усложнить развитие модуля и его зависимостей, так что не следует предпринимать необдуманных действий.

ВАЖНАЯ ИНФОРМАЦИЯ

Продолжая развивать эту мысль, вы могли заметить, что использование transitive должно быть исключением и прибегать к нему следует только при очень определенных обстоятельствах. Наиболее выдающимся является то, что я описал до сих пор: если модуль применяет типы из другого модуля в собственном открытом API (как определено в разделе 3.3), то данное действие должно подразумевать читабельность этого другого модуля с помощью директивы requires transitive.

Другие варианты использования — агрегация, декомпозиция и объединение модулей; все это мы обсудим в подразделе 11.1.5. Прежде я хочу изучить аналогичный вариант применения, который может потребовать другое решение.

До сих пор предполагалось, будто *выставляющий* модуль не может работать без *выставленного*. Что интересно, это не всегда верно. Выставляющий модуль может реализовывать служебные функции, основанные на выставленном, который будет вызывать лишь код, уже использующий выставленный модуль.

Скажем, библиотека *uber.lib* предлагает полезные функции на основе *com.google.common*. Тогда только пользователи Guava могут задействовать *uber.lib*. В таких случаях могут пригодиться необязательные зависимости (см. раздел 11.2).

11.1.4. Когда стоит полагаться на подразумеваемую читабельность

Вы видели, как подразумеваемая читабельность позволяет модулю «передавать» читабельность выставленных зависимостей. Мы обсудили соображения, которые могут повлиять на решение, использовать ли данную функцию. Это была точка зрения разработчика, пишущего выставляющий модуль.

Теперь переключим перспективу и посмотрим с точки зрения *зависимого* модуля: того, на который передается читабельность выставленного модуля. В какой степени он должен опираться на подразумеваемую читабельность? В какой момент он должен вместо этого требовать выставленный модуль?

Как вы видели, когда мы впервые исследовали подразумеваемую читабельность, *java.sql* выставляет свою зависимость от *java.logging*. Возникает вопрос: должны ли модули, использующие *java.sql*, требовать и *java.logging*? Технически такая декларация не нужна и может показаться излишней.

Это верно и для примера с *monitor.peek*, *monitor.observer* и *monitor.observer.alpha*: в конечном решении *monitor.peek* использует типы из обоих модулей, но требует лишь *monitor.observer.alpha*, что подразумевает читабельность *monitor.observer*. Должен ли он также явно требовать *monitor.observer*? И если нет, то только не в этом конкретном примере или вообще никогда?

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Решить, когда полагаться на зависимость, подразумеваемую читабельность модуля или когда запрашивать данный модуль напрямую, возможно, если вернуться к одному из основных обещаний системы модулей: надежной конфигурации (см. подраздел 3.2.1). Использование директив `requires` усиливает надежность кода, делая зависимости явными, и вы можете применить этот принцип здесь, чтобы принять решение, обсудив разные вопросы.

ВАЖНАЯ ИНФОРМАЦИЯ

Зависит ли зависимый модуль от выставленного независимо от выставляющего? Или, другими словами, если зависимый модуль модифицирован так, чтобы больше не использовать выставляющий модуль, может ли он все еще нуждаться в выставленном модуле?

- Если ответ отрицательный, то удаление кода, который использует выставляющий модуль, также удаляет зависимость от выставленного модуля. Можно сказать, что выставленный модуль применялся только на границе между зависимым и выставляющим модулями. В этом случае нет необходимости явно его требовать и вполне достаточно будет полагаться на подразумеваемую читабельность.
- Если, напротив, ответ положительный, то выставленный модуль применяется не только на границе с выставляющим модулем. Соответственно, он должен явно зависеть от директивы `requires`.

На рис. 11.4 показаны эти два варианта.

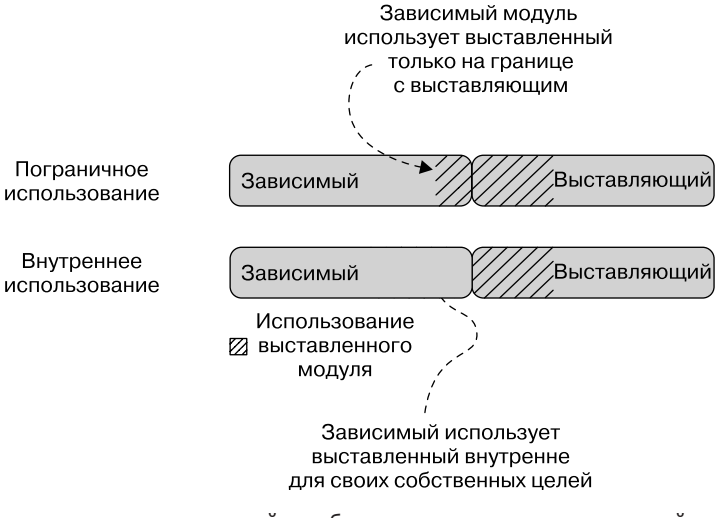


Рис. 11.4. Два случая подразумеваемой читабельности, включающие зависимый, выставляющий и выставленный модули. Там, где два прямоугольника соприкасаются, зависимый модуль использует выставляющий, от которого явно зависит. Оба задействуют выставленный модуль (заштрихованная область). Но степень применения может различаться: зависимый модуль может использовать его только на границе с выставленным *(вверху)* или задействовать типы для реализации своих собственных функций *(внизу)*

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Оглядываясь на пример с *java.sql*, можно ответить на вопрос, основываясь на том, как зависимый модуль, скажем *monitor.persistence*, использует *java.logging*.

- Может потребоваться только чтение *java.logging*, вследствие чего он может вызвать `java.sql.Driver.getParentLogger()`, изменить уровень журнала в регистраторе и покончить с этим. В таком случае его взаимодействие с *java.logging* определяется границей между *monitor.persistence* и *java.sql*, и это наилучшее место для подразумеваемой читабельности.
- В качестве альтернативы *monitor.persistence* может использовать ведение журнала во всем своем коде. Затем типы из *java.logging* появляются во многих местах независимо от `Driver` и больше не могут рассматриваться как определенные границей. В этом случае *monitor.persistence* должен явно требовать *java.logging*.

Подобное сопоставление может быть сделано для примера из приложения *ServiceMonitor*. Использует ли *monitor.peek*, для которого требуется *monitor.observer.alpha*, только типы из *monitor.observer* для создания `ServiceObserver`? Или задействует типы из модуля *monitor.observer* независимо от его взаимодействия с *monitor.observer.alpha*?

11.1.5. Рефакторинг модулей с подразумеваемой читабельностью

На первый взгляд подразумеваемая читабельность выглядит как небольшая функция, которая решает проблему какого-то конкретного случая. Интересно, однако, что она не ограничивается им! Напротив, открывает некоторые полезные методы, помогающие в рефакторинге модулей.

Мотивация для использования этих методов часто состоит в том, чтобы предотвратить изменения в модулях, которые зависят от подвергающихся рефакторингу. При наличии полного контроля над всеми клиентами модуля и при их одновременной компиляции и развертывании вы можете изменить их модульные декларации вместо совершения чего-то более сложного. Но часто это невозможно — например, при разработке библиотеки, — так что нужен способ рефакторинга модулей без нарушения обратной совместимости.

Представление семейств модулей с модулями-агрегаторами

Допустим, в вашем приложении есть пара основных модулей, от которых должен зависеть практически любой другой модуль. Конечно, можно скопировать и вставить необходимые директивы `requires` в каждую модульную декларацию, что довольно утомительно. Вместо этого можно использовать подразумеваемую читабельность для создания так называемого модуля-агрегатора.

Модуль-агрегатор не содержит код и подразумевает читабельность с `requires transitive` на всех своих зависимостях. Он используется для создания связного набора модулей, от которого другие модули могут с легкостью зависеть, просто запрашивая модуль-агрегатор.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Приложение *ServiceMonitor* немного маловато, чтобы оправдать создание в нем модуля-агрегатора; но для примера решим, что *monitor.observer* и *monitor.statistics* являются его основным API. В этом случае можно создать *monitor.core* следующим образом:

```
module monitor.core {
    requires transitive monitor.observer;
    requires transitive monitor.statistics;
}
```

Теперь все остальные модули могут зависеть от *monitor.core* и свободно получать читабельность для *monitor.observer* и *monitor.statistics*. На рис. 11.5 представлен данный пример.

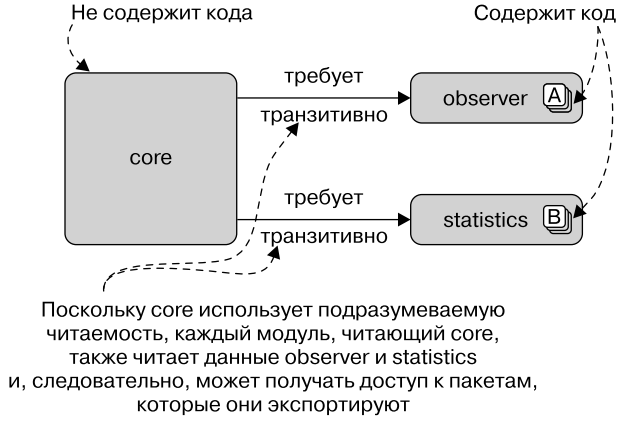


Рис. 11.5. Модуль-агрегатор core *(слева)* не содержит кода и использует директивы requires transitive для ссылки на агрегированные модули observer и statistics *(справа),* которые содержат функциональность. Благодаря подразумеваемой читабельности клиенты модуля-агрегатора могут задействовать API агрегированных модулей

Конечно, нет причин ограничивать агрегацию основными функциями. Каждое семейство модулей, взаимодействующих для реализации функции, — кандидат на получение модуля-агрегатора, который ее представляет.

Но подождите: разве модули-агрегаторы не приводят к тому, что клиенты внутренне используют API модулей, от которых явно не зависят? Это может идти вразрез с тем, что я говорил при обсуждении вариантов, когда стоит полагаться на подразумеваемую читабельность: что ее следует применять на границе с другими модулями. Но я думаю, здесь ситуация несколько иная.

Модули-агрегаторы несут особую ответственность: объединять функциональность связанных модулей в единое целое. Модификация содержимого комплекта — ключевое концептуальное изменение. «Обычная» подразумеваемая читабельность, с другой стороны, часто проявляется между теми модулями, которые не связаны непосредственно (как с *java.sql* и *java.logging*), где подразумеваемый модуль используется более случайно (хотя его модификация все еще может нарушить API; см. подраздел 15.2.4).

Если вы знакомы с терминологией объектно-ориентированного программирования, то можете сравнить это с ассоциацией, агрегацией и композицией (сравнение

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

далеко не идеально, и термины не совсем точно совпадают, но при наличии знаний о терминологии все это интуитивно):

- обычные директивы `requires` создают несложную связь между двумя задействованными модулями;
- использование `requires transitive` превращает это в агрегацию, где один модуль создает другую часть своего API;
- модули-агрегаторы аналогичны композиции в том смысле, что жизненные циклы задействованных модулей связаны — у модуля-агрегатора нет собственного смысла существования. Это, однако, не совсем в точку, поскольку в истинном агрегировании ссылочные модули не имеют своей цели — с модулями-агрегаторами, с другой стороны, она обычно появляется.

Учитывая эти категории, я бы сказал, что наличие открытых зависимостей агрегации должно регулироваться руководящими принципами, представленными в подразделе 11.1.4, тогда как применение открытых зависимостей композиции подходит для любого случая. Чтобы не усложнять ситуацию сверх необходимого, я не буду использовать термины *«агрегация»* и *«композиция»* в остальной части книги; я стану придерживаться *подразумеваемой читабельности* и *модулей-агрегаторов*.

ВАЖНАЯ ИНФОРМАЦИЯ

Наконец, предупреждение: модули-агрегаторы — слабая абстракция! В данном случае они ослабляют сервисы и квалифицированные экспорт и открытие. Последние представлены в разделе 11.3 и подразделе 12.2.2, поэтому здесь я не буду вдаваться в подробности. Достаточно сказать, что они работают, назвав конкретные модули, и поэтому только они могут получить доступ к пакету. Хотя модуль-агрегатор предлагает разработчикам использовать себя вместо своих модулей композиции, экспорт или открытие пакета в модуль-агрегатор не имеет смысла, поскольку он не содержит собственный код, и модули композиции все равно будут видеть сильно инкапсулированный пакет.

Привязка сервисов, как описано в подразделе 10.1.2, также бросает тень на иллюзию модулей-агрегаторов, являющихся идеальными заместителями. Здесь проблема заключается в том, что если модуль композиции предоставляет сервис, то привязка перетянет его в граф модуля, но, разумеется, не в модуль агрегатора (поскольку он не объявляет предоставление данного сервиса) и, следовательно, не создает другие модули композиции. Обдумайте эти случаи перед созданием модулей-агрегаторов.

Рефакторинг модулей путем их разделения

Я уверен, вы оказались в ситуации, когда осознали: то, что когда-то считалось простой функцией, превратилось в более сложную подсистему. Вы улучшали и расширяли ее снова и снова, и теперь она немного запутана. Таким образом, чтобы очистить кодовую базу, нужно реорганизовать ее в более мелкие части, взаимодействующие более определенным образом, сохраняя при этом ее публичный API стабильным.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

На примере приложения ServiceMonitor его статистические операции могли собрать столько кода, что имеет смысл разделить их на несколько меньших подпроектов, скажем Averages, Medians и Percentiles. Пока все нормально; теперь рассмотрим, как это применяется к модулям.

Предположим для начала: простая функция имеет собственный модуль и новое решение будет использовать несколько модулей. Что произойдет с кодом, зависимым от исходного модуля? Если он исчезнет, то система модулей пожалуется на отсутствие зависимостей.

С учетом того, что мы обсудили секунду назад, почему бы не сохранить оригинальный модуль и не превратить его в агрегатор? Это возможно, если *все* экспортированные пакеты исходного модуля теперь экспортируются новыми модулями. (В противном случае зависимость от нового модуля-агрегатора не предоставляет доступ ко всем типам своего прежнего API.)

ВАЖНАЯ ИНФОРМАЦИЯ

Чтобы сохранить зависимости от monitor.statistics без изменений, его можно превратить в модуль-агрегатор. Переместите весь код в новые модули и отредактируйте декларацию модуля monitor.statistics так, чтобы он требовал новый код с ключевым словом transitive:

```
module monitor.statistics {
    requires transitive monitor.statistics.averages;
    requires transitive monitor.statistics.medians;
    requires transitive monitor.statistics.percentiles;
}
```

На рис. 11.6 показано данное разбиение. Это хорошая возможность подтвердить транзитивную природу подразумеваемой читабельности: все модули, зависящие от гипотетического модуля monitor.core, созданного в предыдущем примере, тоже будут читать новые модули statistics, поскольку monitor.core использует requires transitive monitor.statistics и в monitor.statistics прописано requires transitive для новых модулей.

Если хотите, чтобы клиенты заменили свою зависимость от старого модуля более конкретными директивами `requires` для новых модулей, то подумайте, можно ли исключить агрегатор:

```
@Deprecated
module my.shiny.aggregator {
    // ...
}
```

ВАЖНАЯ ИНФОРМАЦИЯ

Предыдущее предупреждение о том, что модули-агрегаторы — слабая абстракция, полностью применимо. Если пользователи задействуют квалифицированный экспорт или открывают модуль-агрегатор, то новые модули от этого не выиграют.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

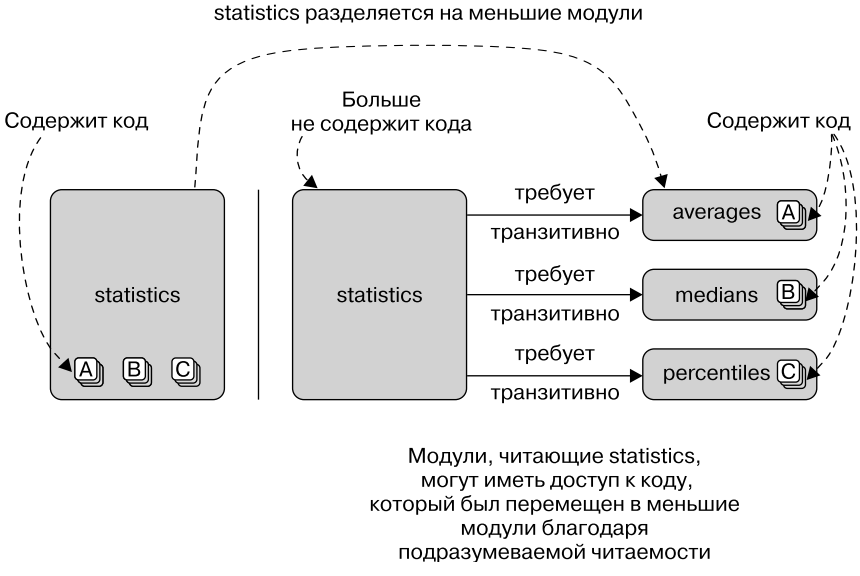


Рис. 11.6. До рефакторинга модуль statistics содержит множество функций *(слева)*. Затем он разбивается на три меньших модуля, которые содержат весь код *(справа)*. Чтобы не требовать изменений в модулях, зависимых от statistics, он не удаляется, а вместо этого превращается в модуль-агрегатор, подразумевающий читабельность модулей, на которые был разбит

11.1.6. Рефакторинг модулей путем их слияния

Можно попробовать объединить несколько модулей — хотя, вероятно, это требуется реже, чем разделение модуля, который вырос из своих границ. Как и раньше, удаление технически бесполезных модулей чревато поломкой на стороне клиента. И, как и прежде, можно использовать подразумеваемую читабельность для решения данной проблемы: оставьте пустые старые модули на прежнем месте и убедитесь, что декларация старого модуля имеет единственную строку — `requires transitive` для нового модуля.

ВАЖНАЯ ИНФОРМАЦИЯ

Работая с приложением ServiceMonitor, вы можете осознать, что наличие модуля на каждую реализацию observer является излишним, и захотеть объединить все модули в monitor.observer. Переместить код из monitor.observer.alpha и monitor.observer.beta в monitor.observer очень просто. Чтобы оставить части приложения, которые напрямую требуют модули реализации без изменений, добавьте их подразумеваемую читабельность в более крупный модуль:

```
@Deprecated
module monitor.observer.alpha {
    requires transitive monitor.observer;
}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
@Deprecated
module monitor.observer.beta {
    requires transitive monitor.observer;
}
```

Эти модули можно увидеть на рис. 11.7. Добавьте в них аннотацию @Deprecated, чтобы побудить пользователей обновить их зависимости.

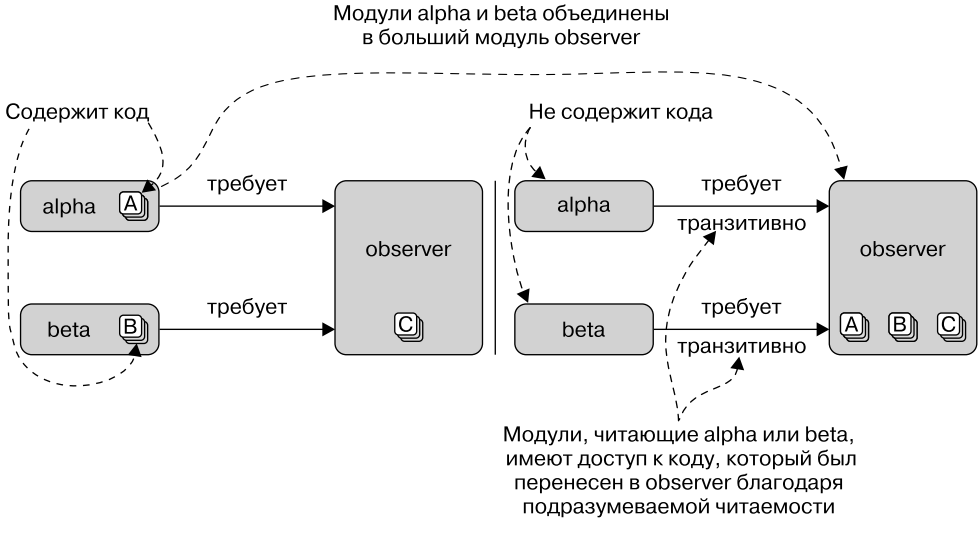


Рис. 11.7. До рефакторинга код модуля observer распределяется между тремя модулями: alpha, beta и observer *(слева)*. После этого вся функциональность находится в observer, и вынесенные модули alpha и beta подразумевают читабельность, чтобы не требовать изменений от своих клиентов *(справа)*

ВАЖНАЯ ИНФОРМАЦИЯ

Однако тщательно продумайте данный подход. Он заставляет клиентов меньших модулей внезапно зависеть от чего-то гораздо большего, чем то, на что они изначально рассчитывали. Кроме того, помните: модули-агрегаторы представляют собой слабую абстракцию.

11.2. Необязательные зависимости

В разделе 3.2 вы увидели, что система модулей использует директивы `requires` для реализации надежной конфигурации, обеспечивая наличие зависимостей во время компиляции и выполнения. Но, как мы обсуждали в конце раздела 2.3, когда впервые рассматривали приложение *ServiceMonitor*, этот подход может быть слишком негибким.

Есть случаи, когда код заканчивается использованием типов, которые не должны присутствовать во время выполнения — они *могут* это делать, но *не обязательно*. В нынешнем виде система модулей требует, чтобы они присутствовали во время

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

запуска (в случае использования директивы `requires`), или вообще не разрешает доступ (когда вы ее не задействуете).

В текущем разделе я покажу пару примеров, в которых данная строгость приводит к проблемам. Затем представлю решение системы модулей: необязательные зависимости. Однако их использование в коде нетривиально, вследствие чего мы внимательно рассмотрим и это. К концу раздела вы сможете писать модули, которые не должны обязательно присутствовать во время выполнения. В ветви `feature-option-dependencies` в репозитории *ServiceMonitor* показано, как применять необязательные зависимости.

11.2.1. Загадка надежной конфигурации

Предположим, что существует расширенная библиотека статистики, содержащая модуль *stats.fancy*, который не может присутствовать в пути модуля для каждого развертывания приложения *ServiceMonitor*. (Причина не имеет значения, но допустим, что она связана с проблемой лицензирования.)

При необходимости добавить код в *monitor.statistics*, который использует типы из модуля *fancy*, для того чтобы это работало, следует определить зависимость от него с помощью директивы `requires`. Но после таких действий система модулей не позволит запустить приложение, если отсутствует *stats.fancy*. На рис. 11.8 показан этот тупик. (Если случай кажется знакомым, то лишь потому, что мы смотрели на него раньше под другим углом. Я подскажу, где именно, когда мы пройдем полный круг через пару минут.)

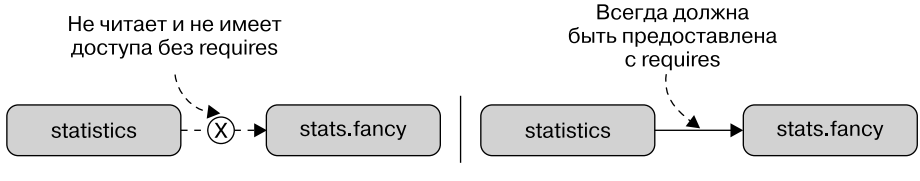


Рис. 11.8. Загадка надежной конфигурации: либо система модулей не предоставляет statistics доступ к stats.fancy, поскольку для statistics не требуется доступ *(слева)*, либо statistics требует доступ, что означает обязательное присутствие stats.fancy для запуска приложения *(справа)*

Другим примером может быть служебная библиотека, назовем ее `uber.lib`, интегрирующаяся с несколькими другими библиотеками. Ее API предлагает функционал на основе других библиотек и предоставляет их типы. Пока это может выглядеть как простой пример подразумеваемой читабельности, как обсуждалось в разделе 11.1, но это можно рассмотреть в другом свете.

Проверим на примере `com.google.common`, с которым интегрируется `uber.lib`. Разработчики, сопровождающие `uber.lib`, могут предположить, что никто из тех, кто еще не использует Guava, никогда не будет называть Guava частью своей библиотеки. Это имеет смысл в определенных случаях. Зачем вам вызывать метод

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

в `uber.lib`, создающий интересный отчет для экземпляра `com.google.common.graph.Graph`, если этого графа у вас нет?

Для `uber.lib` это означает, что она может отлично работать без `com.google.common`. Если Guava делает это в модульном графе, то клиенты могут обращаться к предоставленной части API `uber.lib`. Если нет, то они не будут этого делать, что также подходит для библиотеки. Можно сказать, что `uber.lib` никогда не нуждается в зависимости ради самой себя.

С особенностями системы, которые мы исследовали до сих пор, такие дополнительные отношения не могут быть реализованы. В соответствии с правилами читабельности и доступности из главы 3 `uber.lib` должна требовать, чтобы `com.google.common` компилировался с ее типами и, таким образом, заставлял всех клиентов всегда иметь Guava в пути модуля при запуске их приложения.

Если `uber.lib` интегрируется с несколькими библиотеками, то это заставит клиентов зависеть от них всех, даже если они никогда не будут использовать более одной библиотеки. Это не очень хороший шаг со стороны `uber.lib`, поэтому ее сопровождающие будут искать способ пометить свои зависимости как необязательные во время выполнения. Как показано в следующем подразделе, система модулей может это сделать.

ПРИМЕЧАНИЕ

Инструменты сборки также принимают подобные необязательные зависимости. В Maven для тега зависимости <option> установлено значение true; в Gradle вы перечисляете их после compileOnly.

11.2.2. Модификатор static: маркировка зависимостей как необязательных

Когда модуль должен быть скомпилирован с типами из другого модуля, но не хочет зависеть от него во время выполнения, можно использовать директиву `requires static` для установки этой *необязательной зависимости*. Для двух модулей, *зависимого* и *необязательного*, где декларация первого содержит строку `requires static optional`, система модулей ведет себя по-разному во время компиляции и запуска:

- во время компиляции необязательный модуль должен присутствовать, иначе возникнет ошибка. Во время компиляции необязательный читается зависимым;
- во время запуска необязательный может отсутствовать, и это не вызовет ни ошибки, ни предупреждения. Если он присутствует, то зависимый может его прочитать.

Таблица 11.1 сравнивает это поведение с обычной директивой `requires`. Обратите внимание: хотя система модулей не выдает ошибку, само выполнение может это сделать. Необязательные зависимости делают ошибки выполнения, такие как `NoClassDefFoundError`, гораздо более вероятными, поскольку классы, для которых

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

был скомпилирован модуль, могут отсутствовать. В подразделе 11.2.4 вы увидите код, который готовится к такой возможности.

Таблица 11.1. Сравнение того, как requires и requires static ведут себя во время компиляции и запуска для существующих и отсутствующих зависимостей. Единственная разница заключается в том, как они обрабатывают недостающие зависимости во время запуска (крайний справа столбец)

Директива	Зависимость представлена		Зависимость пропущена	
	Время выполнения	Время запуска	Время выполнения	Время запуска
requires	Читает	Читает	Ошибка	Ошибка
requires static	Читает	Читает	Ошибка	Игнорирует

В качестве примера создадим необязательную зависимость *monitor.statistics* от *stats.fancy*. Для этого возьмем директиву `requires static`:

```
module monitor.statistics {
    requires monitor.observer;
    requires static stats.fancy;
    exports monitor.statistics;
}
```

Если *stats.fancy* отсутствует во время компиляции, то вы получите сообщение об ошибке при компиляции декларации модуля:

```
> monitor.statistics/src/main/java/module-info.java:3:
>     error: module not found: stats.fancy
>         requires static stats.fancy;
>                         ^
> 1 error
```

С другой стороны, во время запуска система модулей не заботится о наличии *stats.fancy*.

Дескриптор модуля для *uber.lib* объявляет все зависимости необязательными:

```
module uber.lib {
    requires static com.google.common;
    requires static org.apache.commons.lang;
    requires static org.apache.commons.io;
    requires static io.vavr;
    requires static com.aol.cyclops;
}
```

Теперь, когда вы знаете, как объявлять необязательные зависимости, осталось ответить на два вопроса.

- При каких обстоятельствах будет присутствовать зависимость?
- Как писать код с необязательной зависимостью?

Далее вы получите ответы на оба и в итоге вы будете готовы использовать эту удобную функцию.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

11.2.3. Разрешение модулей для необязательных зависимостей

Как обсуждалось в подразделе 3.4.1, разрешение модулей — это процесс, который строит модульный граф, разрешая директивы `requires` и учитывая исходный модуль и совокупность обозреваемых модулей. Когда модуль разрешается, все требуемые модули должны быть обозреваемыми. Если они есть, то добавляются в граф; в противном случае возникает ошибка. Чуть позже я написал о графе:

> Важно отметить, что модули, которые не попали в модульный граф во время разрешения, также не будут доступны позже во время компиляции или выполнения.

ВАЖНАЯ ИНФОРМАЦИЯ

Во время компиляции разрешение модулей обрабатывает необязательные зависимости так же, как и обычные. Во время запуска, с другой стороны, директивы requires static в основном игнорируются. Сталкиваясь с директивой, система модулей не пытается выполнить ее, то есть даже не проверяет, является ли указанный модуль обозреваемым.

Как следствие, даже присутствуя в пути модуля (или, раз уж на то пошло, в JDK), модуль не будет добавлен в граф из-за необязательной зависимости. Он попадет в граф только в том случае, если является также обычной зависимостью какого-либо другого модуля, который разрешается, или же если будет добавлен явно с параметром командной строки --add-modules, как описано в подразделе 3.4.3. На рис. 11.9 показаны оба поведения, использована опция для проверки наличия необязательной зависимости.

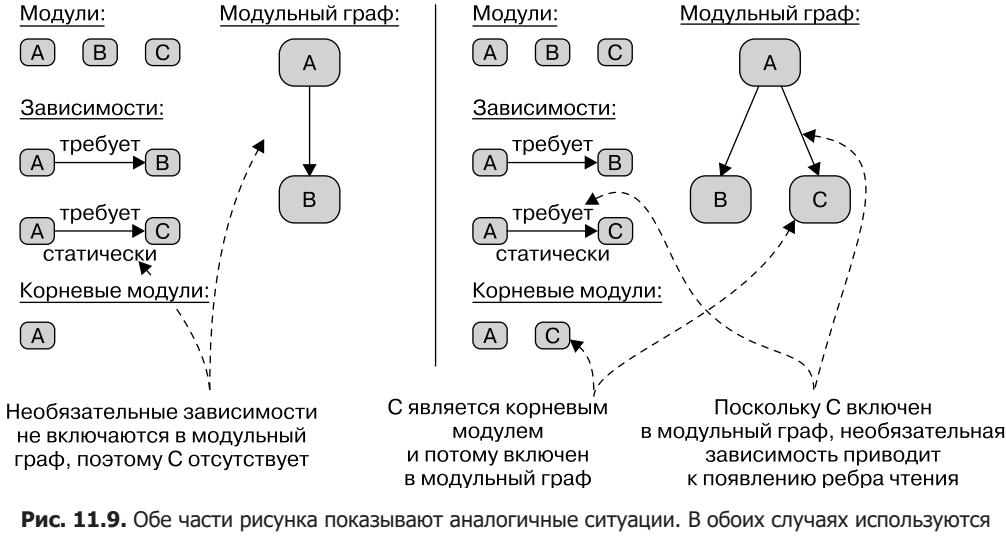


Рис. 11.9. Обе части рисунка показывают аналогичные ситуации. В обоих случаях используются три модуля A, B и C, где A строго зависит от B и не обязательно от C. *Слева* A — начальный модуль, приводящий к графу модулей без C, поскольку необязательные зависимости не разрешены. *Справа* C был введен в граф с помощью параметра командной строки --add-modules, что делает его вторым корневым модулем. Следовательно, он разрешен и читаем модулем A

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Вот где мы пройдем полный круг. Первый раз, когда я упомянул статистическую библиотеку *fancy*, был в разделе, где я объяснял, почему иногда бывает необходимо явно добавить модуль в модульный граф. В частности, я не говорил о дополнительных зависимостях (и это не единственный вариант использования данной опции), но общая идея была такой же, как и сейчас: статистически модуль *fancy* не является строго обязательным и, следовательно, не добавлен автоматически в модульный граф. Если вы хотите, чтобы он был там, то используйте параметр `--add-modules` — и либо назовите конкретный модуль, либо задействуйте `ALL-MODULE-PATH`.

Возможно, вас смутила фраза, что во время разрешения модуля необязательные зависимости «в основном игнорируются». Почему *в основном*? Так, если необязательная зависимость помещается в граф, то система модулей добавляет ребро чтения. Таким образом, в случае нахождения модуля статистики *fancy* в графе (может быть, из-за обычной `requires`, может быть, из-за `--add-modules`) любой модуль, зависимый от него, может прочитать его. Это обеспечивает прямой доступ к его типам.

11.2.4. Написание кода с необязательной зависимостью

Необязательные зависимости требуют немного больше внимания при написании кода для них, поскольку произойдет вот что, если *monitor.statistics* использует типы в *stats.fancy*, но модуль отсутствует во время выполнения:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
    stats/fancy/FancyStats
        at monitor.statistics/monitor.statistics.Statistician
            .<init>(Statistician.java:15)
        at monitor/monitor.Main.createMonitor(Main.java:42)
        at monitor/monitor.Main.main(Main.java:22)
Caused by: java.lang.ClassNotFoundException: stats.fancy.FancyStats
       ... many more
```

Упс. Как правило, вы не хотите, чтобы с кодом происходило такое.

Вообще, когда выполняемый в данный момент код ссылается на тип, JVM проверяет, загружен ли он уже. Если нет, то указывает загрузчику классов сделать это. И если процесс прошел неудачно, появляется ошибка `NoClassDefFoundError`, которая обычно приводит к сбою приложения или по крайней мере к сбою фрагмента выполняемой логики.

Это то, чем славился JAR-ад (см. подраздел 1.3.1). Система модулей хочет преодолеть данную проблему, проверяя заявленные зависимости при запуске приложения. Но при использовании `requires static` вы отказываетесь от данной проверки, что означает вероятное появление `NoClassDefFoundError`. Как можно поступить во избежание этого?

Прежде чем искать решение, необходимо выяснить, действительно ли существует проблема. В случае с `uber.lib` вы ожидаете использования типов из необязательной зависимости только в том случае, если код, вызывающий библиотеку, уже применяет их, то есть загрузка класса выполнена успешно. Другими словами, когда вызывается `uber.lib`, должны присутствовать все необходимые зависимости, иначе вызов будет невозможен. Таким образом, проблемы в конце концов нет и не нужно ничего делать. На рис. 11.10 показан этот случай.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

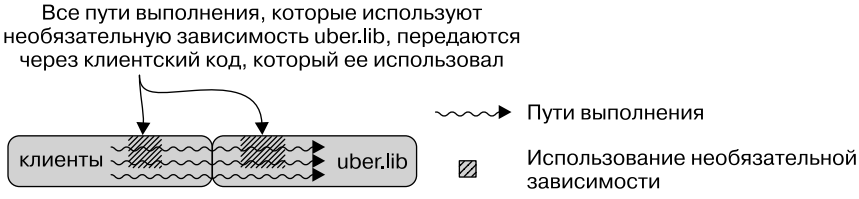


Рис. 11.10. Предполагается, что вызов uber.lib имеет смысл, только когда клиенты уже используют типы из необязательной зависимости. Как следствие, все пути выполнения (волнистые линии), зависящие от необязательной зависимости, доступной для uber.lib (верхние две), уже прошли через клиентский код, который также полагался на эту зависимость (заштрихованные области). Если это не сработало, то uber.lib тоже не заработает

Общий случай отличается от вышеперечисленного, как показано на рис. 11.11. Вполне возможно такое, что модуль с необязательной зависимостью сначала пытается загрузить классы из зависимости, которая может отсутствовать, поэтому риск `NoClassDefFoundError` вполне реален.

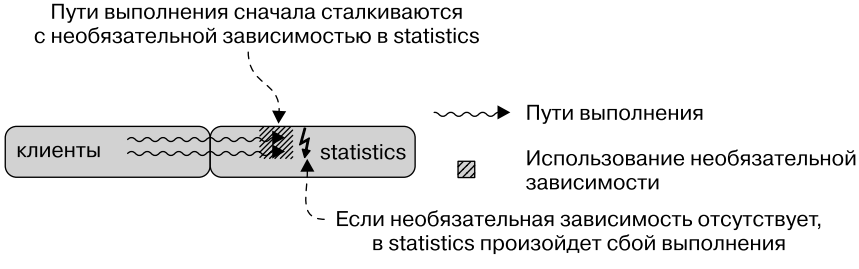


Рис. 11.11. В общем случае нет гарантии, что клиентский код, вызывающий такой модуль, как statistics, уже установил необязательную зависимость. В этом случае пути выполнения (волнистые линии) могут сначала столкнуться с зависимостью в модуле statistics (заштрихованная область) и потерпят неудачу, если необязательная зависимость отсутствует

ВАЖНАЯ ИНФОРМАЦИЯ

Одно из решений — сделать так, чтобы все возможные вызовы модуля с необязательной зависимостью проходили через контрольную точку, прежде чем получить доступ к зависимости. Как показано на рис. 11.12, эта контрольная точка должна оценивать наличие зависимости и отправлять весь код, который поступает на нее, по другому пути выполнения, если той не существует.

Система модулей предлагает API для проверки наличия модуля. Я пока не буду вдаваться в подробности того, как это работает, поскольку у нас нет некоторых предпосылок, необходимых для понимания кода. Поэтому придется дождаться подраздела 12.4.2 для уверенности в том, что можно реализовать такой метод:

```
public static boolean isModulePresent(String moduleName) {
    // ...
}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

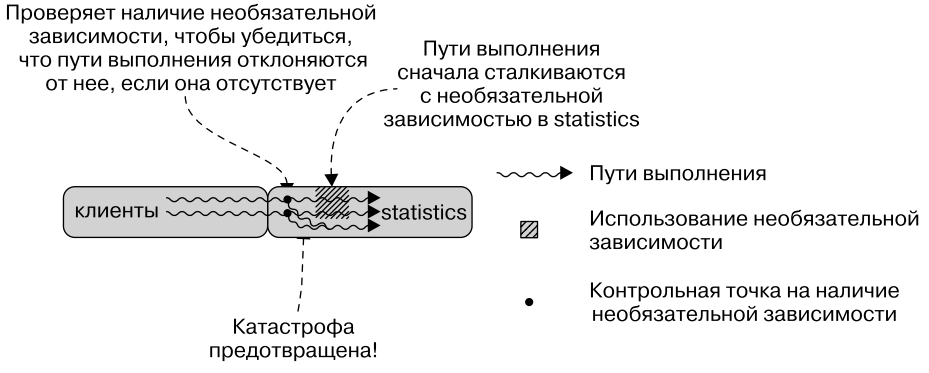


Рис. 11.12. Чтобы гарантировать, что такой модуль с необязательной зависимостью, как statistics, стабилен даже без этой зависимости, требуются контрольные точки. Основываясь на наличии зависимости, код разветвляет пути выполнения (волнистые линии) либо в код, который использует данную зависимость (заштрихованная область), либо в другой код, который этого не делает

Вызов данного метода с аргументом наподобие `"stats.fancy"` вернет `true` при наличии этого модуля. Если он вызывается с именем обычной зависимости (простая директива `requires`), то результат всегда будет истинным, поскольку в противном случае система модулей не позволила бы запускать приложение.

Если метод вызывается с именем необязательной зависимости (директива `requires static`), то результатом будет либо `true`, либо `false`. При наличии необязательной зависимости система модулей установит читабельность, поэтому можно безопасно идти по пути выполнения, который использует типы из модуля. Если необязательная зависимость отсутствует, то выбор такого пути приведет к ошибке `NoClassDefFoundError`, поэтому необходимо найти другой путь.

11.3. Квалифицированный экспорт: ограничение доступа к конкретным модулям

Принимая во внимание, что предыдущие два раздела показывают, как улучшить зависимости, данный раздел вводит механизм, который позволяет более тонко спроектировать API. Как обсуждалось в разделе 3.3, открытый API модуля определяется путем экспорта пакетов с директивами `exports`. В данном случае каждый модуль, считывающий экспортируемый, может получить доступ ко всем открытым типам в этих пакетах во время компиляции и во время выполнения. Это лежит в основе строгой инкапсуляции, что подробно описано в подразделе 3.3.1.

Учитывая то, что мы обсуждали до сих пор, вам придется выбирать между строгой инкапсуляцией пакета или его постоянной доступностью для всех. Обрабатывать варианты использования, которые нелегко вписываются в эту дихотомию, система модулей предлагает с помощью двух менее откровенных способов экспорта пакета: квалифицированного, который мы рассмотрим сейчас; и открытых пакетов, которые представлены в разделе 12.2, поскольку связаны с рефлексией. Как и раньше, я начну с примеров, прежде чем вводить механизм. К концу текущего раздела вы сможете

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

более точно представить API, чем это возможно с обычными директивами `exports`. Посмотрите на ветвь `feature-qualified-exports` в репозитории *ServiceMonitor*, чтобы убедиться, насколько успешными являются экспорты.

11.3.1. Предоставление внутренних API

Лучшие примеры, показывающие, что директивы `exports` могут быть слишком общими, взяты из JDK. Как вы видели в разделе 7.1, лишь один платформенный модуль экспортирует пакет `sun.*` и пару пакетов экспорта `com.sun.*`. Но значит ли это, что все остальные пакеты используются только в модуле, в котором объявлены?

Отнюдь! Многие пакеты распределяются между модулями. Вот некоторые примеры.

- Внутренние компоненты базового модуля *java.base* используются повсеместно. Например, *java.sql* (предоставляющий API подключения к базе данных Java (JDBC)) задействует `jdk.internal.misc`, `jdk.internal.reflect` и `sun.reflect.misc`. Пакеты, относящиеся к безопасности, такие как `sun.security.provider` и `sun.security.action`, применяются *java.rmi* (API удаленного вызова методов (RMI)) или *java.desktop* (инструментарии пользовательского интерфейса AWT и Swing, а также специальные возможности, мультимедиа и JavaBeans API).
- Модуль *java.xml* определяет API Java для обработки XML (JAXP), который включает API потоковой передачи для XML (StAX), простой API для XML (SAX) и API объектной модели документов (DOM) W3C. Шесть его внутренних пакетов (в основном с префиксами `com.sun.org.apache.xml` и `com.sun.org.apache.xpath`) используются *java.xml.crypto* (API для криптографии XML).
- Многие модули JavaFX обращаются к внутренним пакетам *javafx.graphics* (в основном `com.sun.javafx.*`), который, в свою очередь, использует `com.sun.javafx.embed.swing` из *javafx.swing* (объединяющий JavaFX и Swing). Тот, в свою очередь, задействует семь внутренних пакетов из *java.desktop* (например, `sun.awt` и `sun.swing`), которые...

Я мог бы продолжать, но уже уверен, что вы поняли мою точку зрения. Однако возникает вопрос: как JDK распределяет эти пакеты среди своих модулей, не экспортируя их всеми остальными?

Хотя JDK, безусловно, предлагает самый сильный вариант использования для более целенаправленного механизма экспорта, он не является единственным. Такая ситуация возникает каждый раз, когда набор модулей хочет поделиться функциональностью между ними, не раскрывая ее. Это может произойти с библиотекой, фреймворком или даже подмножеством модулей из более крупного приложения.

Это симметрично проблеме сокрытия служебных классов до введения системы модулей. Как только служебный класс должен быть доступен в разных пакетах, он должен стать публичным; но до Java 9 это означало, что весь код, работающий в той же JVM, мог получить к нему доступ. Теперь вы столкнулись с тем, что хотите скрыть пакет утилит, но, как только он должен быть доступен в разных модулях, он должен быть экспортирован и, таким образом, доступен всем модулям, работающим

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

в одной JVM, — по крайней мере с помощью механизмов, которые вы использовали до сих пор. На рис. 11.13 показана данная симметрия.

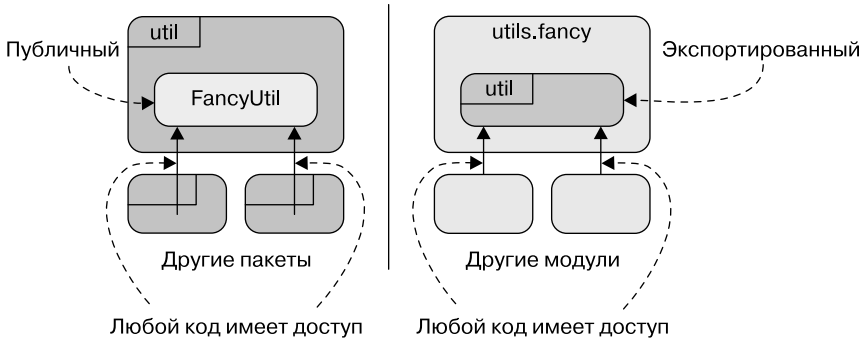


Рис. 11.13. *Слева:* ситуация до Java 9, когда сразу после того, как тип становится публичным (например, FancyUtil в пакете util), к нему может обращаться любой другой код. *Справа:* аналогичная ситуация с модулями, но на более высоком уровне, когда после экспорта пакета (например, util в utils.fancy) он становится доступным для всех других модулей

11.3.2. Экспорт пакетов в модули

Директиву `exports` можно уточнить с помощью размещения за ней `${`*модули*`}`, где `${`*модули*`}` — разделенный запятыми список имен модулей (заместители не допускаются). Для модулей, указанных в директиве `exports to`, пакет будет точно так же доступным, как и с обычной директивой `exports`. Для всех остальных модулей пакет будет настолько сильно инкапсулирован, словно бы `exports` не было вообще. Эта ситуация показана на рис. 11.14.

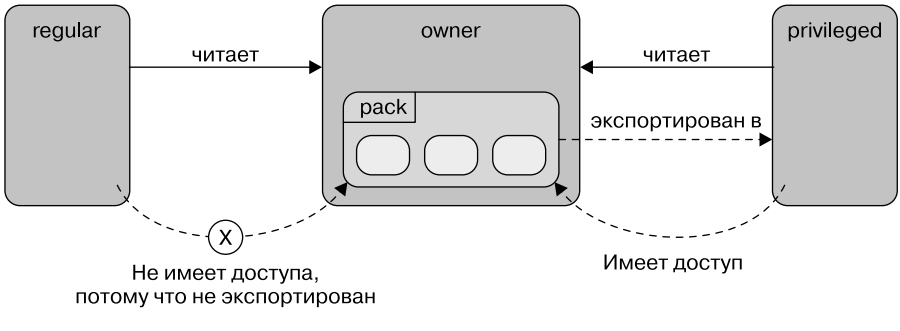


Рис. 11.14. Модуль owner использует квалифицированный экспорт, чтобы пакет pack был доступен только для модуля privileged. Для privileged он так же доступен, как если бы использовался обычный экспорт; но другие модули, такие как regular, не могут получить доступ к нему

В качестве гипотетического примера, скажем, все реализации *observer* в приложении *ServiceMonitor* должны совместно использовать некий служебный код. Первый вопрос: куда поместить эти типы? Все *observer* уже зависят от *monitor.observer*, по-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

скольку он содержит интерфейс `ServiceObserver`, который они реализуют, так почему бы не поместить его туда? Хорошо, они в конечном итоге помещаются в пакет `monitor.observer.utils`.

Теперь начинается самое интересное. Вот декларация модуля в *monitor.observer*, который экспортирует новый пакет только в модули реализации:

```
module monitor.observer {
    exports monitor.observer;
    exports monitor.observer.utils
        to monitor.observer.alpha, monitor.observer.beta;
}
```

В то время как `monitor.observer` экспортируется во все модули, `monitor.observer.utils` будет доступен лишь модулям *monitor.observer.alpha* и *monitor.observer.beta*.

Этот пример демонстрирует две интересные детали.

- Модули, в которые экспортируется пакет, могут зависеть от экспортируемого модуля, создавая цикл. Развивая эту мысль, можно понять, что если подразумеваемая читабельность не используется, то это *должно* иметь место: как иначе модуль, в который экспортируется пакет, прочитал бы экспортируемый?
- Всякий раз, когда новая реализация хочет использовать утилиты, необходимо изменить модуль API, чтобы он предоставлял доступ к этому новому модулю. Хотя предоставление экспортируемому модулю контроля над тем, какой код может получить доступ к пакетам, — один из основных моментов квалифицированного экспорта, он все же может быть громоздким.

В качестве примера из реальной жизни я хотел бы показать вам квалифицированные экспорты, которые объявляет *java.base*, но их 65, так что это было бы чрезмерным. Вместо этого рассмотрим дескриптор модуля *java.xml* с помощью `java --describe-module java.xml` (как описано в подразделе 5.3.1):

```
> module java.xml@9.0.4
# все, кроме квалифицированного экспорта, опущено
> qualified exports com.sun.org.apache.xml.internal.utils
>     to java.xml.crypto
> qualified exports com.sun.org.apache.xpath.internal.compiler
>     to java.xml.crypto
> qualified exports com.sun.xml.internal.stream.writers
>     to java.xml.ws
> qualified exports com.sun.org.apache.xpath.internal
>     to java.xml.crypto
> qualified exports com.sun.org.apache.xpath.internal.res
>     to java.xml.crypto
> qualified exports com.sun.org.apache.xml.internal.dtm
>     to java.xml.crypto
> qualified exports com.sun.org.apache.xpath.internal.functions
>     to java.xml.crypto
> qualified exports com.sun.org.apache.xpath.internal.objects
>     to java.xml.crypto
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Это показывает, что *java.xml* позволяет *java.xml.crypto* и *java.xml.ws* использовать некоторые из его внутренних API.

Теперь, когда вы знаете о квалифицированном экспорте, я могу раскрыть небольшую загадку, которую мы оставили в подразделе 5.3.6 при анализе журналов системы модулей. Там вы видели такие сообщения:

```
> Adding read from module java.xml to module java.base
> package com/sun/org/apache/xpath/internal/functions in module java.xml
>     is exported to module java.xml.crypto
> package javax/xml/datatype in module java.xml
>     is exported to all unnamed modules
```

Я не объяснил, почему в журнале говорится об экспорте *в* модуль, но с учетом того, что мы обсудили секунду назад, сейчас это должно быть понятно. Как вы видели в недавнем примере, *java.xml* экспортирует `com.sun.org.apache.xpath.internal.functions` в *java.xml.crypto*, что в точности соответствует второму сообщению. Третье сообщение экспортирует *javax.xml.datatype* во «все безымянные модули». Это выглядит немного странно, но таков способ системы модулей сказать, что пакет экспортируется без дополнительной квалификации и, следовательно, доступен для каждого модуля, читающего *java.xml*, включая безымянный модуль.

ВАЖНАЯ ИНФОРМАЦИЯ

Наконец, две небольшие заметки о компиляции.

Если модуль, который объявляет квалифицированный экспорт, компилируется, а целевой модуль отсутствует в совокупности обозреваемых модулей, то компилятор выдаст предупреждение. Это не ошибка, поскольку целевой модуль упоминается, но не является обязательным.

Запрещается использовать пакет в директивах exports и exports to. Если бы присутствовали обе директивы, то последние были бы фактически бесполезны, поэтому такая ситуация интерпретируется как ошибка реализации и, следовательно, приводит к ошибке компиляции.

11.3.3. Когда использовать квалифицированный экспорт

Квалифицированный экспорт позволяет модулям совместно использовать пакет между собой, не делая его доступным для всех других модулей в той же JVM. Это делает квалифицированные экспорты полезными для библиотек и фреймворков, состоящих из нескольких модулей и желающих обмениваться кодом без возможности применения его клиентами. Они также пригодятся для больших приложений, которые хотят ограничить зависимости от определенных API.

Квалифицированный экспорт можно рассматривать как поднятие строгой инкапсуляции от защиты типов в артефактах до защиты пакетов в наборах модулей. Это показано на рис. 11.15.

К примеру, вы разрабатываете модуль. Когда следует отдавать предпочтение квалифицированному экспорту вместо неквалифицированного экспорта? Чтобы ответить на этот вопрос, мы должны сосредоточиться на основном преимуществе

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

квалифицируемого экспорта: на контроле над тем, кто использует API. Вообще говоря, это становится тем важнее, чем дальше данный пакет от его клиентов.

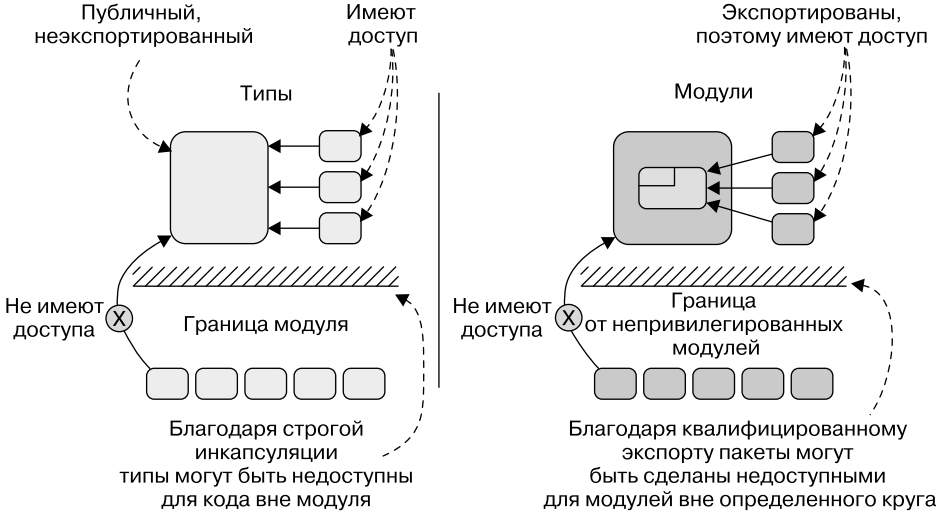


Рис. 11.15. *Слева:* как открытый тип в неэкспортированном пакете может быть доступен другим типам в том же модуле, но не типам из других модулей. *Справа:* аналогичная ситуация, но на более высоком уровне, когда квалифицированные экспорты используются для того, чтобы сделать пакет в одном модуле доступным для определенного набора модулей, сохраняя при этом его недоступность для непривилегированных модулей

Предположим, у вас есть приложение небольшого или среднего размера, состоящее из нескольких модулей (не считая зависимостей), которое сопровождается небольшой командой, компилируется и развертывается одновременно. В этом случае сравнительно легко контролировать, какой модуль использует тот или иной API; и если что-то пойдет не так, то это легко исправить, поскольку все находится под контролем. В данном сценарии преимущества квалифицированного экспорта оказывают незначительное влияние.

На другом конце спектра находится JDK, который используется буквально каждым Java-проектом в мире и уделяет особое внимание обратной совместимости. Наличие кода «там», зависящего от внутреннего API, может вызывать проблему и быть трудноисправимым, поэтому было бы здорово контролировать, кто и к чему получает доступ.

Наиболее очевидная граница, разделяющая эти две крайности, заключается в том, можно ли свободно изменять клиентов пакета. Если можно, поскольку вы разрабатываете модуль и все его клиентские модули, то обычный экспорт — это хороший выбор. Если нельзя, так как вы сопровождаете библиотеку или фреймворк, то лишь API, который нужно использовать для клиентов и который вы хотите сопровождать, должен экспортироваться без квалификации. Все, кроме этого, особенно внутренние утилиты, должно экспортироваться только в ваши модули.

Данная граница размыта в более крупных проектах. Если большая кодовая база в течение многих лет сопровождается большой командой, то можно *технически* изме-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

нить всех клиентов, когда это станет необходимым из-за изменения API, но данный процесс может быть болезненным. В таких случаях использование квалифицированного экспорта не только предотвращает случайные зависимости от внутренних пакетов, но и документирует, для каких именно клиентов был разработан API.

11.3.4. Экспорт пакетов в командной строке

Предположим, использование внутренних API не было предусмотрено (или, скорее, не предполагалось) на момент написания? Что, если у кода *должен быть* абсолютный доступ к типам, которые содержащий модуль не экспортирует, квалифицированно или нет? Будь система модулей непреклонна в отношении этих правил, многие приложения не компилировались бы и не запускались на Java 9+. Но если бы был простой способ обойти строгую инкапсуляцию, она вряд ли была бы «строгой», теряя при этом свои преимущества. Срединная позиция нашлась путем определения параметров командной строки, которые можно использовать в качестве аварийного люка, но они слишком громоздки, чтобы применять их в качестве повсеместного исправления.

В дополнение к директиве `export to` существует параметр командной строки с точно таким же эффектом, который можно применить к командам компилятора и выполнения: с помощью `--add-exports ${`*`модуль`*`}/${`*`пакет`*`}=${`*`модули_доступа`*`}` система модулей экспортирует `${`*`пакет`*`}` из *`$модуль`* во все модули, перечисленные в списке через запятую `${`*`модули_доступа`*`}`>. Если среди них есть `ALL-UNNAMED`, то код из безымянного модуля тоже может прочитать этот пакет.

Применяются обычные правила доступности, представленные в разделе 3.3, — для доступа модуля к типу с помощью параметра `--add-exports` должны быть выполнены следующие условия:

- тип должен быть публичным;
- тип должен находиться в `${`*`пакет`*`}`;
- модуль, указанный в `${`*`модули_доступа`*`}`, должен читать *`${модуль}`*.

Для примеров `--add-exports` вернитесь к подразделам 7.1.3 и 7.1.4, где мы использовали его для доступа к внутренним API платформенных модулей во время компиляции и выполнения. Как и другие параметры командной строки, требование наличия `--add-exports` для чего-то большего, чем экспериментальные примеры, является проблемой удобства сопровождения; см. раздел 9.1, чтобы узнать подробности.

Резюме

- Подразумеваемая читабельность.
  - С помощью директивы `requires transitive` модуль заставляет своего клиента читать требуемый модуль, даже если явно не зависит от него. Это позволяет модулю использовать типы из зависимостей в его API, не прибегая к необходимости вручную запрашивать эти зависимости на клиентских модулях. Как следствие, модуль становится мгновенно используемым.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

  - Модуль должен полагаться лишь на то, что транзитивная зависимость является неявно читаемой, если он использует ее исключительно на границе с соответствующей прямой зависимостью. Как только модуль начинает использовать транзитивную зависимость для реализации собственной функциональности, он должен сделать ее прямой зависимостью. Это гарантирует, что объявление модуля отражает истинный набор зависимостей и делает модуль более устойчивым к рефакторингу, который может удалить транзитивную зависимость.
  - Подразумеваемая читабельность может использоваться при перемещении кода между модулями, поскольку модули, ранее содержавшие код, подразумевают читабельность тех, которые содержат его сейчас. Это позволяет клиентам получать доступ к коду, от которого они зависят, не требуя от них изменения своих модульных дескрипторов, поскольку они все равно в конечном итоге читают модуль, содержащий код. Сохранение подобной совместимости особенно интересно для библиотек и фреймворков.
- Необязательные зависимости.
  - С помощью директивы `requires static` модуль отмечает зависимость, которую система модулей проверит на предмет того, присутствует ли та во время компиляции, но в то же время она может отсутствовать во время выполнения. Это позволяет писать код для модулей, не заставляя клиентов всегда иметь эти модули в своем приложении.
  - Во время запуска модули, требующиеся *только* с помощью директив `requires static`, не добавляются в модульный граф, даже если являются обозреваемыми. Вместо этого их нужно добавить вручную с помощью `--add-modules`.
  - При написании кода с необязательными зависимостями необходимо убедиться, что ни один из путей выполнения не может завершиться сбоем из-за отсутствия зависимости, поскольку это серьезно подорвет удобство использования модуля.
- Квалифицированный экспорт.
- С помощью директивы `export to` модуль делает пакет доступным только для названных модулей. Это третий и наиболее целевой вариант между инкапсуляцией пакета и обеспечением его доступности для всех.
- Экспорт в определенные модули позволяет совместно использовать код в наборе привилегированных модулей, не делая его общедоступным API. Это уменьшает поверхность API библиотеки или фреймворка, тем самым улучшая удобство сопровождения.
- С помощью параметра командной строки `--add-exports` можно экспортировать во время компиляции и выполнения пакеты, которые разработчики модуля рассматривали как внутренние API. С одной стороны, это поддерживает выполнение кода, зависящего от этих внутренних компонентов, с другой стороны, создает собственные проблемы с сопровождением.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

12 Рефлексия в модульном мире

В этой главе

- Открытие пакетов и модулей для рефлексии.
- Объединение модулей и рефлексии.
- Альтернативы API рефлексии.
- Анализ и изменение свойств модуля.

Если вы работаете с Java-приложением, то велика вероятность использования Spring, Hibernate, JAXP, GSON или тому подобных программ. Что значит «тому подобных»? Фреймворков, которые используют API рефлексии Java для проверки кода, поиска аннотаций, создания объектов или вызова методов. Благодаря рефлексии они могут делать все это, не прибегая к компиляции кода.

Кроме того, API рефлексии позволяет фреймворкам получать доступ к закрытым классам и членам. Он обладает сверхспособностями за пределами того, что возможно для скомпилированного кода, который находится вне границ пакета, если классы или члены не являются публичными. Дело в том, что с модулями рефлексия больше не работает «из коробки».

Скорее наоборот: рефлексия утратила свои сверхспособности и привязана к тем же правилам доступности, что и скомпилированный код. Она может получить доступ только к открытым членам открытых классов в экспортированных пакетах. Эти фреймворки, с другой стороны, используют рефлексию над полями и методами, часто не являющимися публичными, и над классами, которые вы, возможно,

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

не захотите экспортировать, поскольку они не выступают частью API модуля. Что делать в таком случае? Вот об этом как раз и говорится в данной главе!

Чтобы извлечь максимальную пользу из этой главы, вам следует:

- иметь общее представление о том, как работает рефлексия (в противном случае приложение Б поможет вам освоиться быстрее);
- знать, что каждый раз, помещая аннотацию куда-либо, вы помечаете этот класс для фреймворка, чтобы рефлексировать его (см. листинг 12.1 с несколькими примерами);
- понимать правила доступности (как представлено в разделе 3.3).

Листинг 12.1. Фрагменты кода для стандартов и фреймворков, основанных на рефлексии

```
// JPA
@Entity
@Table(name = "user")
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    @Column(name = "id", updatable = false, nullable = false)
    private Long id;

    @Column(name = "title", nullable = false)
    private String title;

    // [...]

}

// JAXB
@XmlRootElement(name = "book")
@XmlAccessorType(XmlAccessType.FIELD)
public class Book {

    @XmlElement
    private String title;

    @XmlElement
    private String author;

    // [...]

}

// SPRING
@RestController
public class BookController {
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
    @RequestMapping(value = "/book/{id}", method = RequestMethod.GET)
    @ResponseBody
    public Book getBook(@PathVariable("id") long id) {
        // [...]
    }

    // [...]

}
```

С учетом вышеперечисленного вы узнаете, почему директивы `exports` не позволят вам уйти далеко вперед, если вы хотите разрешить рефлексивный доступ к своим модулям (см. раздел 12.1) и что можно сделать вместо этого (см. раздел 12.2). (Обратите внимание: это применимо только к явным модулям — если код запускается из пути к классам, то он не инкапсулирован, так что вам не нужно беспокоиться о чем-либо.)

Но эта глава — о чем-то большем, чем «просто» подготовка модулей к рефлексии: в ней рассматривается и другая сторона и обсуждается, как обновить рефлексивный код, а также альтернативы и дополнения к API рефлексии (см. раздел 12.3). Она заканчивается советами, как можно использовать слои для динамической загрузки модулей во время выполнения (см. раздел 12.4). (Эти два раздела написаны для разработчиков, у которых уже был опыт применения чего-либо из вышеперечисленного до Java 9, поэтому они требуют большего знакомства с рефлексией и загрузчиками классов, чем остальная часть главы.)

К концу главы вы будете знать все о подготовке проекта к рефлексии в модульном мире, независимо от того, рефлексируется ли он, или рефлексирует другой код. Вы также сможете использовать рефлексию для динамической загрузки кода во время выполнения, например, для реализации приложения на основе плагинов.

12.1. Почему директивы exports не подходят для рефлексии

Прежде чем мы приступим к тому, как наилучшим образом подготовить код к рефлексии, имеет смысл обсудить, почему механизм, обсуждаемый до сих пор для представления классов, директива `exports` (см. раздел 3.3), не подходит. На то есть три причины:

- крайне сомнительно, что классы, предназначенные для использования с такими платформами, должны быть частью открытого API модуля;
- экспорт этих классов в выбранный модуль может связать модуль с реализацией вместо стандарта;
- экспорт не поддерживает глубокую рефлексию частных полей и методов.

Мы рассмотрим каждую причину по очереди после обсуждения того, как все работало до системы модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

12.1.1. Врываемся в немодульный код

Допустим, вы успешно перенесли свое приложение в Java 9+, как описано в главах 6 и 7. Однако еще не модулировали его, поэтому все еще запускаете его из пути к классам. В таком случае рефлексия кода продолжает работать как в Java 8.

Основанные на рефлексии фреймворки будут регулярно создавать и изменять экземпляры классов, получая доступ к закрытым типам и членам. Хотя невозможно скомпилировать код с видимыми в пакете или закрытыми элементами, рефлексия позволяет применять их после того, как они становятся доступными. В листинге 12.2 показан гипотетический устойчивый фреймворк, который использует рефлексию для создания объекта и назначения идентификатора частному полю.

Листинг 12.2. Использование рефлексии

```
Class<?> type = ...
Constructor<?> constructor = entityType.getConstructor();
constructor.setAccessible(true);
Object entity = constructor.newInstance();
Field id = entity.getDeclaredField("id");
id.setAccessible(true);
id.set(entity, 42);
```

Что-нибудь нужное фреймворку для доступа к этому классу

Делает возможно приватный конструктор и поле доступными для следующих вызовов

12.1.2. Принудительная публикация внутренних типов

ВАЖНАЯ ИНФОРМАЦИЯ

В соответствии с правилами доступности, описанными в разделе 3.3, типы должны быть публичными, а в экспортированном пакете — доступными. Это верно и для рефлексии; так что, не используя директиву exports, вы получите ошибку, подобную показанной ниже:

```
> Exception in thread "main" java.lang.IllegalAccessException:
>   class p.X (in module A) cannot access class q.Y (in module B)
>   because module B does not export q to module A
>       at java.base/....Reflection.newIllegalAccessException
>       at java.base/....AccessibleObject.checkAccess
>       at java.base/....Constructor.newInstance
```

Похоже, это указывает на то, что вам нужно сделать классы, к которым Spring, Hibernate и т. д. должны были получить доступ, публичными и экспортировать пакеты, содержащие их. Указанное действие добавляет их к публичному API модуля, и, поскольку мы до сих пор считаем данные типы внутренними, это серьезное решение.

Если вы пишете небольшой сервис с несколькими тысячами строк кода, разбитых на несколько модулей, то это может не показаться проблемой. В конце концов, вряд

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ли что-то может сильно запутаться в API и связях модулей. Но это также не тот сценарий, когда стоит использовать модули.

С другой стороны, если вы работаете с большей кодовой базой с номерами строк кода в шесть или семь цифр, разбитых на десятки или сотни модулей, над которыми работают дюжина или более разработчиков, то все выглядит совсем по-другому. В данном сценарии экспорт пакета дает другим разработчикам четкий сигнал о том, что можно использовать эти классы вне модуля и они специально предназначены для применения за пределами модуля — в конце концов, затем и нужен экспорт.

Но поскольку отправной точкой для этого исследования было то, что вы по какой-то причине предпочли *не* публиковать эти классы, вы, очевидно, ценили их инкапсуляцию. Было бы довольно иронично, если бы система модулей заставила вас пометить некий объект как поддерживаемый, который не нужно было даже делать доступным, — и это *ослабило бы* инкапсуляцию.

12.1.3. Квалифицированный экспорт создает связь с определенными модулями

Теперь вернитесь к разделу 11.3 и подумайте об использовании квалифицированного экспорта для получения гарантии того, что лишь один модуль может получить доступ к этим внутренним компонентам. Да, похвально, что вы все схватываете на лету, — это действительно может решить проблему, которую я только что описал.

Однако это *может* привести к новой проблеме. Подумайте о JPA и его различных реализациях, таких как Hibernate и EclipseLink. В зависимости от выбранного стиля, вы, возможно, усердно работали над предотвращением прямых зависимостей от выбранной реализации, поэтому не будете с нетерпением ждать, чтобы жестко запрограммировать их в объявлении модуля с `exports … to concrete.jpa.implementation`. Если вы полагаетесь на квалифицированный экспорт, то тем не менее данный путь не подходит.

12.1.4. Нет поддержки глубокой рефлексии

Необходимость делать типы, рассматриваемые как детали реализации, доступными для другого кода — это плохо. И становится хуже.

Скажем, вы установили директиву `exports` (квалифицированную или нет), чтобы позволить выбранному фреймворку получить доступ к классам. Хотя часто можно использовать основанные на рефлексии фреймворки только с открытыми членами, это не всегда так и не всегда является лучшим подходом. Напротив, обычно стоит полагаться на глубокую рефлексию частных полей или непубличных методов, чтобы не раскрывать связанные с фреймворком детали для остальной части кода. (В листинге 12.1 показан ряд примеров, а в листинге 12.2 — то, как `setAccessible` применяется для доступа к внутренним компонентам.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

К счастью, в целом — но, к несчастью, в этом сценарии — создание публичного типа и экспорт его пакета не предоставляют доступ к непубличным членам. Если среда попытается использовать их, вызвав setAccessible, то получится ошибка, подобная этой:

```
> Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
>   Unable to make field q.Y.field accessible:
>   module B does not "opens q" to module A
>       at java.base/java.lang.reflect.AccessibleObject.
    checkCanSetAccessible
>       at java.base/java.lang.reflect.AccessibleObject.
    checkCanSetAccessible
>       at java.base/java.lang.reflect.Field.checkCanSetAccessible
>       at java.base/java.lang.reflect.Field.setAccessible
```

Если вы действительно хотите пойти по такому пути, то вам придется сделать публичными все члены с рефлексивным доступом, что значительно ухудшает сделанное ранее заключение «это ослабляет инкапсуляцию».

Подводя итог, можно выделить следующие недостатки использования директив `exports` для кода, который в первую очередь должен применяться рефлексивно:

- разрешается доступ только к публичным членам, что часто требует обнародования деталей реализации;
- другим модулям можно компилировать код с этими открытыми классами и членами;
- квалифицированный экспорт может связать вас с реализацией вместо спецификации;
- директива `exports` помечает пакет как часть открытого API модуля.

За вами остается выбор, какой из этих четырех недостатков считать худшим. Я же выбираю последний.

12.2. Открытые пакеты и модули: предназначены для использования с рефлексией

Теперь, когда мы установили, насколько `exports` непригодны для обеспечения доступности кода для библиотек на основе рефлексии, какую альтернативу предлагает система модулей?

Ответ — директива `opens`, и это первое, что мы собираемся рассмотреть (см. подраздел 12.2.1), прежде чем вводить ее квалифицированный вариант (сродни `exports … to`, см. подраздел 12.2.2). Чтобы вы могли убедиться в выборе правильного инструмента для работы, мы также тщательно сравним результаты экспорта и открытия модуля (см. подраздел 12.2.3). И последнее, но не менее важное: «кувалда» для предоставления рефлективного доступа — открытые модули (см. подраздел 12.2.4).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

12.2.1. Открытие пакетов для доступа в режиме выполнения

Определение: директива opens

Пакет можно *открыть*, добавив *директиву* `opens ${пакет}` к декларации модуля. Во время компиляции открытые пакеты сильно инкапсулированы: нет разницы между тем, открыты они или нет. Во время выполнения открытые пакеты полностью доступны, включая непубличные классы, методы и поля.

Модуль *monitor.persistence* использует Hibernate, поэтому открывает пакет с сущностями, чтобы можно было их рефлексировать:

```
module monitor.persistence {
    requires hibernate.jpa;
    requires monitor.statistics;

    exports monitor.persistence;
    opens monitor.persistence.entity;
}
```

Это позволяет Hibernate работать с такими классами, как `StatisticsEntity` (листинг 12.3). Поскольку пакет не экспортируется, другие модули *ServiceMonitor* не могут случайно скомпилировать код для содержащихся в нем типов.

Листинг 12.3. Выдержки из StatisticsEntity, которые рефлексируются Hibernate

```
@Entity
@Table(name = "stats")
public class StatisticsEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    @ManyToOne
    @JoinColumn(name = "quota_id", updatable = false)
    private LivenessQuotaEntity totalLivenessQuota;

    private StatisticsEntity() { }

    // [...]

}
```

Hibernate может добавлять значения в эти приватные поля

Hibernate также имеет доступ к приватному конструктору

Как вы можете заметить, `opens` была создана специально для случая рефлексии и ведет себя совершенно иначе, чем `exports`:

- предоставляет доступ всем членам, что не влияет на решения разработчика относительно видимости;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- запрещает компиляцию кода в открытых пакетах и разрешает доступ только во время выполнения;
- помечает пакет как предназначенный для использования фреймворком на основе рефлексии.

Помимо явных технических преимуществ перед `exports` для этого конкретного случая применения, я снова нахожу последний пункт самым важным: с помощью директивы `opens` вы четко сообщаете в коде, что данный пакет предназначен не для общего использования, а лишь для доступа конкретного инструмента. При желании можно даже включить данный инструмент, открыв пакет только для его модуля. Читайте дальше, чтобы узнать, как это сделать. Как объясняется в подразделе 5.2.3, если нужно предоставить доступ к таким ресурсам, как конфигурации или мультимедийные файлы, которые находятся в ваших пакетах, то необходимо открыть и их.

12.2.2. Открытие пакетов для определенных модулей

Директива `opens`, о которой мы только что говорили, позволяет всем модулям рефлексировать открытый пакет. Это аналогично тому, как `exports` позволяет всем модулям получать доступ к экспортированному пакету. Обе директивы: и `exports`, и `opens` — могут быть ограничены определенными модулями (см. раздел 11.3).

> **Определение: выборочное открытие**
>
> Директиву `opens` можно уточнить, добавив к ней `to` `${`*модули*`}`, где `${`*модули*`}` — список имен модулей, разделенных запятыми (заместители не допускаются). Для модулей, указанных в директиве `opens to`, пакет будет точно таким же доступным, как и с обычной директивой `opens`. Для всех остальных модулей пакет будет настолько сильно инкапсулирован, как если бы `opens` вообще не было.

Чтобы сделать инкапсуляцию еще более сильной, *monitor.persistence* может открыть свой пакет сущностей лишь для Hibernate:

```
module monitor.persistence {
    requires hibernate.jpa;
    requires monitor.statistics;
    exports monitor.persistence;
    // предполагая, что Hibernate был явным модулем
    opens monitor.persistence.entity
        to hibernate.core;
}
```

В случаях, когда спецификации и реализации разделены (например, JPA и Hibernate), может показаться немного странным упоминание реализации в декларации модуля. В подразделе 12.3.5 рассматривается данная мысль: в итоге это потребуется до тех пор, пока стандарты не будут обновлены с учетом системы модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В подразделе 12.2.3 мы обсудим, когда можно применять выборочные открытия, но прежде, чем сделать это, официально представим параметр командной строки, который мы использовали в разделе 7.1.

Определение: --add-opens

Опция `--add-opens ${`*`модуль`*`}/${`*`пакет`*`}=${`*`рефлексирующий_модуль`*`}` открывает `${`*`пакет`*`}` из `${`*`модуль`*`}` в `${`*`рефлексирующий_модуль`*`}`. Таким образом, код в `${`*`рефлексирующий_модуль`*`}` может получить доступ ко всем типам и членам, как открытым, так и приватным, в `${`*`пакет`*`}`, но другие модули не могут.

При установке `${`*`читающий_модуль`*`}` в `ALL-UNNAMED` весь код из пути класса или, точнее, из безымянного модуля (см. раздел 8.2) может получить доступ к этому пакету. При переходе на Java 9+ вы всегда будете использовать этот заместитель — только после запуска собственного кода в модулях можно ограничить открытые пакеты конкретными модулями.

Если вас интересует пример, то отыщите его в конце подраздела 7.1.4.

Поскольку `--add-opens` связано с рефлексией, чистой концепцией времени выполнения, то имеет смысл только для команды `java`. Что интересно, оно все-таки доступно на `javac`, но приводит к выдаче предупреждения:

```
> warning: [options] --add-opens has no effect at compile time
```

Одно из моих лучших предположений о том, почему `javac` не отвергает полностью `--add-opens`, заключается в том, что это позволяет совместно использовать один и тот же файл аргументов с флагами командной строки, относящимися к системе модулей, между компиляцией и запуском.

ПРИМЕЧАНИЕ

Что такое файлы аргументов? Можно поместить аргументы компилятора и JVM в файл и добавить их в команду с помощью javac @file-name и java @file-name. Подробности см. в документации Java: mng.bz/K1ZK).

12.2.3. Экспорт и открытие пакетов

Директивы `exports` и `open` имеют несколько общих черт:

- делают содержимое пакета доступным за пределами модуля;
- имеют квалифицированный вариант `${`*`модули`*`}`, который дает доступ только к перечисленным модулям;
- имеют параметры командной строки для `javac` и `java`, которые можно использовать для обхода строгой инкапсуляции в случае необходимости.

Их различия состоят в том, когда и к чему они дают доступ:

- экспортированные пакеты предоставляют доступ к открытым типам и членам во время компиляции, что делает их идеальными для определения публичных API, которые могут использоваться другими модулями;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- открытые пакеты предоставляют доступ ко всем типам и членам (включая непубличные), но только во время выполнения, что делает их хорошо подходящими для предоставления основанным на рефлексии средам доступа к коду, который в противном случае считается внутренним для модуля.

В табл. 12.1 показано краткое сравнение вышеперечисленных свойств. Вы также можете вернуться к табл. 7.1, чтобы увидеть, как она связана с получением доступа к внутренним API с помощью `--add-exports` и `--add-opens`.

Таблица 12.1. Сравнение того, когда и к чему инкапсулированные, экспортированные и открытые пакеты дают доступ

Доступ	**Время компиляции**		**Время выполнения**	
Класс или член	Публичный	Непубличный	Публичный	Непубличный
Инкапсулированный пакет	✘	✘	✘	✘
Экспортированный пакет	✔	✘	✔	✘
Открытый пакет	✘	✘	✔	✔

Вы можете задаться вопросом, можно ли и как комбинировать директивы `exports` и `opens`, а также квалифицированные и неквалифицированные варианты. Ответ прост — так, как вам нравится:

- ваши сущности Hibernate являются публичными API? Используйте `exports` и `opens`;
- хотите, чтобы только некоторые из модулей приложения имели доступ во время компиляции к контекстам Spring? Задействуйте `exports … to` и `opens`.

Может не быть очевидного варианта использования для каждой из четырех возможных комбинаций (и я бы даже поспорил, что вам стоит разработать свой код так, чтобы он не понадобился), но будьте уверены: если вы столкнетесь с ним, то сможете организовать эти директивы соответственно.

Когда дело доходит до того, должны ли открытия быть ограничены конкретными модулями, я считаю, что это часто не стоит дополнительных усилий. Пусть квалифицированные экспорты и важный инструмент, предотвращающий непреднамеренное применение коллегами и пользователями внутренних API (подробности см. в подразделе 11.3.3), целевая аудитория для квалифицированных открытий — платформы, полностью независимые от вашего кода. Независимо от того, открываете ли вы пакет только в Hibernate, Spring не будет зависеть от него. Если ваш проект рефлексирует много собственного кода, то все может выглядеть иначе; но в остальном я по умолчанию открываю модули — без квалификации.

12.2.4. Открытие модулей: закрытие рефлексии

Наконец, если у вас есть большой модуль с множеством пакетов, которые можно рефлексировать, то может быть утомительно открывать каждый из них по отдельности. Хотя нет никакого символа подстановки, например `opens com.company.*`, существует нечто близкое к нему.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Определение: открытый модуль

При помещении *ключевого слова* `open` перед `module` в декларации модуля создается *открытый модуль*:

```
open module ${имя_модуля} {
   requires ${имя_модуля};
   exports ${имя_пакета};
   // открытие не разрешается
}
```

Открытый модуль открывает все содержащиеся в нем пакеты, как если бы каждый из них использовался в директиве `opens`. Следовательно, нет смысла открывать дополнительные пакеты вручную, поэтому компилятор не принимает директивы `opens` в открытом модуле.

В качестве альтернативы открытию `monitor.persistence.entity` может быть открыт модуль *monitor.persistence*:

```
open module monitor.persistence {
    requires hibernate.jpa;
    requires monitor.statistics;

    exports monitor.persistence;
}
```

Как видите, открытые модули — это просто удобство, позволяющее не открывать десятки пакетов вручную. В идеале вы никогда не окажетесь в таком положении, поскольку ваши модули не такие большие. Сценарий с подобным количеством открытых пакетов более вероятен во время модуляризации, при превращении большого JAR в большой модуль перед его разбиением. Вот почему `jdeps` может генерировать декларации для открытых модулей (см. подраздел 9.3.2).

12.3. Рефлексия модулей

В разделах 12.1 и 12.2 мы рассмотрели, как можно подготовить код для рефлексии, чтобы такие фреймворки, как Hibernate и Spring, могли получить к нему доступ. Поскольку большинство Java-приложений используют подобные платформы, с этим сценарием вы будете сталкиваться регулярно.

Теперь мы собираемся перейти на другую сторону и рефлексировать модульный код. Полезно знать, как все работает, так что вы можете улучшить понимание API рефлексии. Но, поскольку с написанием кода рефлексии большинство разработчиков встречается редко, есть вероятность, что вы не будете делать это регулярно. Следовательно, данный раздел — скорее обсуждение заслуживающих внимания аспектов рефлексии модулей и их кода, чем подробное введение во все связанные темы и API.

Сначала мы рассмотрим, почему не нужно менять код рефлексии для работы с модульным кодом (см. подраздел 12.3.1) и почему можно перейти на более современный API (см. подраздел 12.3.2). Затем перейдем к самим модулям, имеющим заметное представление в API рефлексии, который можно использовать для их запроса

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

(см. подраздел 12.3.3) и даже изменения (см. подраздел 12.3.4). В заключение более подробно рассмотрим, как можно модифицировать модуль, чтобы позволить другим модулям рефлексировать доступ к нему (см. подраздел 12.3.5).

12.3.1. Обновление рефлективного кода для модулей (или нет)

Прежде чем отправиться на новые территории, я хочу обновить ваши знания о рефлексии, рассказав об изменениях, вызванных системой модулей. Хотя было бы хорошо понять, как рефлексия имеет дело с читабельностью и доступностью, вы обнаружите отсутствие необходимости сильно менять код. Более важно то, что вы информируете своих пользователей об их действиях при создании модулей.

Ничего не нужно делать для читабельности

Одна вещь, о которой я неоднократно говорил, состоит в том, что рефлексия связана с теми же правилами доступности, что и статический доступ (см. раздел 3.3). Прежде всего, это означает, что для получения возможности доступа к коду в другом модуле один модуль должен читать второй. Вообще говоря, модульный граф не будет настроен таким образом — Hibernate обычно не читает модули приложения.

ВАЖНАЯ ИНФОРМАЦИЯ

Описанное выше звучит так, словно рефлексирующий модуль должен добавить ребро чтения от него до рефлексированного модуля, и действительно, для этого есть API (см. подраздел 12.3.4). Но поскольку для рефлексии всегда требуется данное ребро, его добавление в любом случае приведет к неизбежному шаблону, вследствие чего API рефлексии делает это внутренне. Таким образом, разработчикам не нужно беспокоиться о читабельности.

Ничто не может быть сделано для доступности

Следующее препятствие на пути доступа к коду — необходимость его экспорта или открытия. Как подробно обсуждалось в разделе 12.2, это действительно проблема, хотя вы, как автор библиотеки рефлексии, мало что можете сделать с ней. Либо владелец модуля подготовил пакет, открыв или экспортировав его, либо нет.

ВАЖНАЯ ИНФОРМАЦИЯ

Система модулей не ограничивает видимость: вызовы типа Class::forName или рефлексии для получения ссылок на конструкторы, методы и поля завершаются успешно. Доступность ограничена: если доступ не предоставляется рефлексированным модулем, то вызов конструктора или метода, доступ к полю и вызовы AccessibleObject::setAccessible завершатся неудачей с InaccessibleObjectException.

`InaccessibleObjectException` расширяет `RuntimeException`, делая его *непроверенным исключением*, поэтому компилятор не заставит вас его перехватить. Но убе-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

дитесь в том, что вы все делаете правильно, и на данной операции тоже — таким образом, вы можете предоставить пользователям максимально полезное сообщение об ошибке. В листинге 12.4 есть пример.

> **Определение: AccessibleObject::trySetAccessible**
>
> Если вы предпочитаете проверять доступность, не вызывая исключение, то вам нужен метод `AccessibleObject::trySetAccessible`, добавленный в Java 9. По сути, он делает то же самое, что и `setAccessible(true)`: пытается сделать базовый элемент доступным, но использует возвращаемое значение, чтобы указать, сработал ли он. Если доступ был предоставлен, то возвращает `true`, в противном случае возвращается `false`. Листинг 12.4 показывает это в действии.

Листинг 12.4 Три способа обработки недоступного кода

```
private Object constructWithoutExceptioHandling(Class<?> type)
        throws ReflectiveOperationException {
    Constructor<?> constructor = type.getConstructor();
    constructor.setAccessible(true);
    return constructor.newInstance();
}
```

Этот вызов может вызвать исключение InaccessibleObjectException, которое явно не обрабатывается — пользователю остается решить проблему

```
private Object constructWithExceptionHandling(Class<?> type)
        throws ReflectiveOperationException, FrameworkException {
    Constructor<?> constructor = type.getConstructor();
    try {
        constructor.setAccessible(true);
    } catch (InaccessibleObjectException ex) {
        throw new FrameworkException(createErrorMessage(type), ex);
    }
    return constructor.newInstance();
}
```

Здесь исключение преобразуется в специфичное для фреймворка с дополнительным сообщением об ошибке, объясняя контекст, в котором это произошло

```
private Object constructWithoutException(Class<?> type)
        throws ReflectiveOperationException, FrameworkException {
    Constructor<?> constructor = type.getConstructor();
    boolean isAccessible = constructor.trySetAccessible();
    if (!isAccessible)
        throw new FrameworkException(createErrorMessage(type));
    return constructor.newInstance();
}
```

Используя trySetAccessible, исходное исключение можно предотвратить, но тем не менее генерируется новое исключение, специфичное для фреймворка

```
private String createErrorMessage(Class<?> type) {
    return "When doing THE FRAMEWORK THING, accessing "
        + type + "'s parameterless constructor failed "
        + "because the module does not open the containing package. "
        + "For details see https://framework.org/java-modules";
}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Помимо уверенности в том, что вы правильно обрабатываете случай, когда доступ не может быть предоставлен, больше ничего нельзя сделать. Это делает обновление вашего проекта для системы модулей более сложной коммуникационной задачей, чем технической: пользователи должны знать, к каким пакетам может понадобиться доступ проекту и что с этим делать. Ваша документация — очевидное место для их обучения.

> **Выделенная страница JPMS**
>
> Немного не по теме: я рекомендую создать отдельную страницу в документации для вопроса о том, как подготовить модуль для применения в вашем проекте. Чем более он сфокусирован, тем более вероятно, что пользователи найдут его, поэтому не хороните его в уже гигантском документе. Затем распространите данный ресурс по всему миру, в том числе в своем Javadoc и сообщении об исключении для неудачного доступа.

12.3.2. Использование обработчика переменных вместо рефлексии

Java 9 представила новый API, называемый *обработчиком переменных* (расширяя дескрипторы методов Java 7, для которых немногие разработчики находят применение). Он сосредоточен вокруг класса `java.lang.invoke.VarHandle`, экземплярами которого являются строго типизированные ссылки на переменные: например, поля (хотя этим дело не ограничивается). В нем рассматриваются варианты использования из таких областей, как рефлексия, параллелизм и хранение данных вне кучи. По сравнению с API рефлексии он обеспечивает большую безопасность типов и лучшую производительность.

Методы и переменные — универсальные сложные функции, которые имеют мало общего с системой модулей, поэтому я не буду их здесь официально описывать. Даже если вы только время от времени пишете код, использующий рефлексию, вам непременно следует изучить их — для простого примера см. листинг 12.5. Однако есть один особенно интересный аспект, который я хочу обсудить более подробно: как обработчики переменных могут использоваться для предоставления доступа к внутренним компонентам модуля.

Листинг 12.5. Использование VarHandle для получения значения поля

```
Object object = // ...
String fieldName = // ...

Class<?> type = object.getClass();
Field field = type.getDeclaredField(fieldName);

Lookup lookup = MethodHandles.lookup();
VarHandle handle = lookup.unreflectVarHandle(field);
handle.get(object);
```

Даны объект и название поля…

Это типичный код рефлексии для получения типа и поля

Lookup и VarHandle — части API обработчика метода/переменной, который основан на поиске

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Вы уже видели, что API рефлексии требует, чтобы пользователь открыл некоторые пакеты, но у рефлексирующей платформы нет возможности выразить это в коде. Пользователь либо знает это, основываясь на своем понимании системы модулей, либо должен изучить ее, прочитав документацию, — ни то ни другое не является наиболее надежным способом выражения требования. Что, если код фреймворка может это упростить?

Методы и переменные предлагают инструмент для этого. Посмотрите еще раз на листинг 12.5 — видите вызов метода `MethodHandles.lookup()`? Он создает экземпляр `Lookup`, который, помимо прочих привилегий и информации, захватывает права доступа вызывающей стороны.

Это значит, что весь код, независимо от того, к какому модулю он принадлежит, получающий этот конкретный экземпляр `lookup`, может глубоко рефлексировать те же классы, что и код, который создал поиск (рис. 12.1). Таким образом, модуль может захватывать свои права доступа к своим собственным внутренним компонентам и передавать их другим модулям.

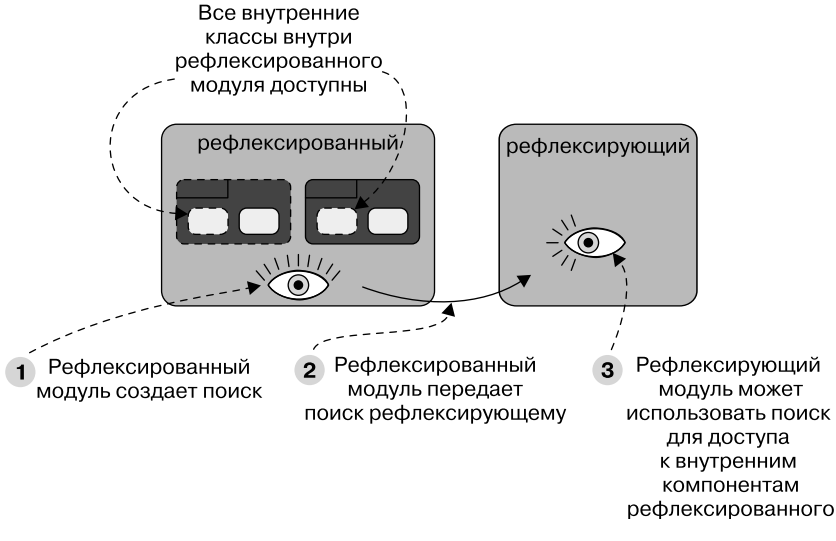


Рис. 12.1. Рефлексированный модуль создает поиск и передает его рефлексирующему, а тот, в свою очередь, затем может использовать его для доступа к тем же классам и членам, к которым может получить доступ рефлексированный модуль, — к ним относятся внутренние элементы рефлексированного модуля

Ваш рефлексирующий код может применять это, требуя, чтобы пользователь передавал ему объекты поиска; например, при загрузке платформы. Когда пользователям приходится вызывать метод, принимающий один или несколько экземпляров `Lookup`, они обязаны читать документы, чтобы узнать о действиях, которые должны совершить. Затем они создают экземпляр в каждом модуле, к которому необходимо получить доступ, и передают его вам, а вы задействуете их для доступа к внутренним компонентам их модуля. В листинге 12.6 показано, как это работает.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Листинг 12.6. Использование VarHandle для доступа к значению поля с приватным поиском

```
Lookup lookup = // ...
Object object = // ...
String fieldName = // ...

Class<?> type = object.getClass();
Field field = type.getDeclaredField(fieldName);

Lookup privateLookup = MethodHandles
    .privateLookupIn (type,lookup);
VarHandle handle = privateLookup.unreflectVarHandle(field);
handle.get(object);
```

Этот поиск был создан в модуле, владеющем объектом

Создав приватный поиск из предоставленного пользователем, можно получить доступ к внутренним объектам объекта из другого модуля

Интересный факт о поисках заключается в том, что они могут передаваться между модулями. В ситуации стандартного разделения по сравнению с разделением реализаций, как в случае JPA и его поставщиков, пользователь может передавать запросы на методы начальной загрузки JPA, которые затем могут передавать их в Hibernate, EclipseLink и т. п. Я думаю, это довольно удобный способ реализации поиска:

- пользователи знают, что им нужно что-то делать, поскольку методы начальной загрузки требуют экземпляров `Lookup` (в отличие от требования открывать пакеты, которое не может быть выражено в коде);
- нет необходимости изменять декларации модулей (в отличие от директив `opens`);
- стандарты могут передавать поиск в реализации, таким образом не заставляя пользователей ссылаться на реализацию в декларациях кода или модуля (это возможно и для открытых пакетов, как объясняется в подразделе 12.3.5).

На этом мы завершаем обсуждение использования рефлексии или обработчиков переменных для доступа к типам, инкапсулированным в модули. Теперь обратимся к самим модулям и посмотрим, какую информацию о них можно получить.

12.3.3. Анализ свойств модуля с помощью рефлексии

Если вы когда-нибудь пытались проанализировать JAR во время выполнения, то наверняка обнаружили, что это неудобно. Данный факт восходит к фундаментальной интерпретации того, чем являются JAR: простыми контейнерами (см. раздел 1.2). Java не распознает их как первоклассные сущности наподобие пакетов и типов, поэтому у него нет представления во время выполнения, которое рассматривает их как нечто большее, чем просто Zip-файлы.

Основное изменение системы модулей заключается в согласовании интерпретации Java JAR-файлов с нашей версией как единицами кода, имеющими имена, зависимости и явные API. Помимо всего, что мы обсуждали до сих пор в данной книге, это следует отнести к API рефлексии, где должны быть представлены модули, в отличие от JAR, но как пакеты и типы. И действительно, они есть.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Определение: типы Module и ModuleDescriptor

Java 9 ввел новый тип `java.lang.Module`, который представляет модуль во время выполнения. Экземпляр `Module` позволяет:

- анализировать имя модуля, аннотации, директивы `exports/opens` и применение сервисов;
- давать доступ к ресурсам, которые содержит модуль (см. раздел 5.2);
- модифицировать модуль, экспортируя и открывая пакеты или добавляя ребра чтения и использование сервисов (если модифицирующий код находится в том же модуле).

Некоторые из этих фрагментов информации доступны только для одинаково нового типа `java.lang.module.ModuleDescriptor`, возвращаемого `Module::getDescriptor`.

Один из способов получить экземпляры `Module` — вызов `getModule` для любого экземпляра `Class`, возвращающего, что неудивительно, модуль, которому принадлежит этот класс. В листинге 12.7 показано, как анализировать модуль, запрашивая `Module` и `ModuleDescriptor`; вывод для некоторых примеров модулей показан в листинге 12.8.

Листинг 12.7. Анализ модуля путем запроса Module и ModuleDescriptor

```
public static String describe(Module module) {
    String annotations = Arrays
        .stream(module.getDeclaredAnnotations())
        .map(Annotation::annotationType)
        .map(Object::toString)
        .collect(joining(", "));
    ModuleDescriptor md = module.getDescriptor();
    if (md == null)
        return "UNNAMED module { }";

    return ""
        + "@[" + annotations + "]\n"
        + md.modifiers() + " module " + md.name()
        + " @ " + toString(md.rawVersion())
        + " {\n"
        + "\trequires " + md.requires() + "\n"
        + "\texports " + md.exports() + "\n"
        + "\topens " + md.opens() + "\n"
        + "\tcontains " + md.packages() + "\n"
        + "\tmain " + toString(md.mainClass()) + "\n"
        + "}";
}

private static String toString(Optional<?> optional) {
    return optional.isPresent()
            ? optional.get().toString()
            : "[]";
}
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Листинг 12.8. Результат вызова describe(Module) из листинга 12.7

```
> @[]
> [] module monitor @ [] {
>     requires [
>         monitor.observer,
>         monitor.rest
>         monitor.persistence,
>         monitor.observer.alpha,
>         mandated java.base (@9.0.4),
>         monitor.observer.beta,
>         monitor.statistics]
>     exports []
>     opens []
>     contains [monitor]
>     main monitor.Main
> }
>
> @[]
> [] module monitor.persistence @ [] {
>     requires [
>         hibernate.jpa,
>         mandated java.base (@9.0.4),
>         monitor.statistics]
>     exports [monitor.persistence]
>     opens [monitor.persistence.entity]
>     contains [
>         monitor.persistence,
>         monitor.persistence.entity]
>     main []
> }
>
> @[]
> [] module java.logging @ 9.0.4 {
>     requires [mandated java.base]
>     exports [java.util.logging]
>     opens []
>     contains [
>         java.util.logging,
>         sun.util.logging.internal,
>         sun.net.www.protocol.http.logging,
>         sun.util.logging.resources]
>     main []
> }
>
> @[]
> [] module java.base @ 9.0.4 {
>     requires []
>     exports [... lots ...]
>     opens []
>     contains [... lots ...]
>     main []
> }
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Некоторые методы `ModuleDescriptor` возвращают информацию, относящуюся к другим модулям: например, какие модули требуются или какие пакеты модулей экспортируются и открываются. Это просто имена модулей в виде строк, а не фактические экземпляры модулей. В то же время многие методы `Module` требуют ввода этих экземпляров. Таким образом, вы получаете строки, но нужно вставить модули — как преодолеть данный разрыв? Как показано в подразделе 12.4.1, ответ — слои.

12.3.4. Изменение свойств модуля с помощью рефлексии

В дополнение к анализу свойств модуля также можно использовать `Module` для их изменения, вызывая эти методы:

- `addExports` экспортирует пакет в модуль;
- `addOpens` открывает пакет для модуля;
- `addReads` позволяет одному модулю читать другой;
- `addUses` заставляет модуль задействовать сервис.

Просматривая их, вы можете задаться вопросом, почему доступны экспорт или открытие пакетов модуля. Разве это не идет вразрез с сильной инкапсуляцией? Разве мы не провели весь раздел 12.2, обсуждая действия, которые должен сделать владелец модуля, чтобы подготовиться к рефлексии, поскольку рефлексирующий код не может просто вломиться в нужный модуль?

ВАЖНАЯ ИНФОРМАЦИЯ

Вот в чем дело: эти методы чувствительны к вызывающим, то есть ведут себя по-разному в зависимости от кода, который их вызывает. Чтобы вызов был успешным, он должен поступать либо из модифицируемого, либо из безымянного модуля. В противном случае произойдет сбой с IllegalCallerException.

Возьмите следующий код в качестве примера:

```
public boolean openJavaLangTo(Module module) {
    Module base = Object.class.getModule();
    base.addOpens("java.lang", module);
    return base.isOpen("java.lang", module);
}
```

При копировании в метод `main`, который выполняется из пути класса (вследствие чего выполняется в безымянном модуле), это работает нормально, и метод возвращает `true`. Если, с другой стороны, он запускается из какого-либо именованного модуля (*open.up* в следующем примере), то завершается ошибкой:

```
> Exception in thread "main" java.lang.IllegalCallerException:
>     java.lang is not open to module open.up
>     at java.base/java.lang.Module.addOpens(Module.java:751)
>     at open.up/open.up.Main.openJavaLangTo(Main.java:18)
>     at open.up/open.up.Main.main(Main.java:14)
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Вы можете заставить его работать (снова), вставляя код в модифицируемый модуль, а именно *java.base*, с помощью `--patch-module` (см. подраздел 7.2.4):

```
$ java
    --patch-module java.base=open.up.jar
    --module java.base/open.up.Main
> WARNING: module-info.class ignored in patch: open.up.jar
> true
```

Вот и все: окончательное значение `true` — это возвращаемое значение из `openJavaLangTo`, вызываемое произвольным платформенным модулем.

Динамическое изменение свойств ваших модулей не является чем-то, чем вы будете заниматься на регулярной основе, даже если разрабатываете фреймворк на основе рефлексии. Так зачем тогда все это рассказывать? Затем, что, как вы увидите в следующем подразделе, здесь скрыта одна интересная деталь: можно открывать пакеты других модулей при определенных обстоятельствах.

12.3.5. Пересылка открытых пакетов

Я сказал, что только модуль может открыть один из своих пакетов с помощью `Module:addOpens`, но это не совсем так. Если пакет модуля уже открыт для набора других модулей, то все эти модули тоже могут открыть данный пакет. Другими словами, модули с рефлективным доступом к пакету могут открывать этот пакет для *других* модулей. Что это значит?

Еще раз подумайте о JPA. Возможно, вы вздрогнули в подразделе 12.2.2, когда выглядело так, будто нужно открыть пакет безоговорочно или для модуля, выполняющего реальную рефлексию, поскольку в случае JPA это будет означать нечто вроде этого:

```
module monitor.persistence {
    requires hibernate.jpa;
    requires monitor.statistics;

    exports monitor.persistence;
    // предполагая, что Hibernate был явным модулем
    opens monitor.persistence.entity
        to hibernate.core;
}
```

Не лучше ли открыть JPA вместо конкретной реализации? Именно это стало возможным благодаря тому, что модули с рефлективным доступом открывают пакеты для других модулей! Таким образом, загрузочный код JPA может открыть все пакеты в Hibernate, даже те, которые имеют рефлективный доступ.

Таким образом, несмотря на то что лишь модуль может добавлять экспорт пакетов, считывать ребра и использовать сервисы, правило открытия пакетов смягчено и все модули, для которых был открыт пакет, могут открывать его для других модулей. Чтобы основанные на рефлексии фреймворки могли использовать это, они, конечно, должны быть знакомы с системой модулей и обновлять свой код. Однако

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

в случае технологий JEE это может занять некоторое время, если только Eclipse не примет более быстрый цикл выпуска Jakarte EE (от Java SE 8 до Java EE 8 прошло более трех лет).

Теперь, когда мы решили, как рефлексировать, проанализировать и изменить отдельный модуль, мы можем перевести его на следующий уровень или, вернее, *слой*, как вы увидите в следующем разделе, и работать со всем модульным графом.

12.4. Динамическое создание диаграмм модулей со слоями

В разделе 12.3 мы сосредоточились на отдельных модулях: как рефлексировать модульный код и как анализировать и изменять свойства одного модуля. В этом разделе мы расширяем наши возможности и рассматриваем целые графы модулей.

До сих пор мы оставляли создание модульных графов компилятору или JVM, которые генерируют их перед началом своей работы. С этого момента граф представляет собой почти неизменную сущность, которая не позволяет добавлять или удалять модули. Это хорошо для многих заурядных приложений, однако есть такие, которые требуют большей гибкости. Подумайте о серверах приложений или приложениях на основе плагинов. Им нужен динамический механизм, который позволяет им загружать и выгружать классы во время выполнения.

В качестве примера предположим, что приложение *ServiceMonitor* предлагает конечную точку или графический интерфейс, с помощью которого пользователь может указать, что нужно наблюдать дополнительный сервис. Это можно сделать, создав экземпляр соответствующей реализации `ServiceObserver`, но что, если данная реализация поступит от модуля, который был неизвестен во время запуска? Тогда он (и его зависимости) должен быть загружен динамически во время выполнения.

До системы модулей подобные контейнерные приложения использовали загрузчики классов для динамической загрузки и выгрузки, но было бы неплохо, если бы они, подобно компилятору и JVM, могли также подняться на более высокий уровень абстракции и вместо этого работать с модулями? К счастью, система модулей позволяет именно это, представляя концепцию *слоев*. Первое, что нужно сделать, — познакомиться со слоями, включая тот, который существовал все время без вашего ведома (см. подраздел 12.4.1). Следующий шаг — анализ слоев (см. подраздел 12.4.2) перед динамическим созданием собственных во время выполнения (см. подраздел 12.4.3).

Обратите внимание: написание кода, который имеет дело со слоями, встречается даже реже, чем использование API рефлексии. Вот простая лакмусовая бумажка: если вы никогда не создавали загрузчик классов, то вряд ли будете применять слои в ближайшее время. Соответственно, данный раздел дает базовое представление, поэтому вы знаете свой путь, но не обращаете внимания на подробности. Тем не менее, вероятно, вы можете увидеть нечто новое и в конечном итоге прийти к парочке новых идей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

12.4.1. Что такое слои

Определение: модульный слой

Модульный слой включает в себя полностью разрешенный граф именованных модулей, а также загрузчик (-и) классов, используемый для загрузки классов модулей. С каждым загрузчиком классов связан безымянный модуль (доступный через `ClassLoader::getUnnamedModule`). Кроме того, слой ссылается на один или несколько родительских слоев — модули в слое могут читать модули в слоях-предках, но не наоборот.

Все, что мы обсуждали до сих пор о разрешении и взаимосвязях между модулями, происходит *внутри одного модульного графа*. Со слоями можно составлять столько графов, сколько захочется, поэтому концептуально слои добавляют третье измерение к двухмерной концепции модульных графов. Родительские слои определяются при создании слоя и не могут быть изменены впоследствии, поэтому возможность создавать циклические слои отсутствует. На рис. 12.2 показан граф модуля со слоями.

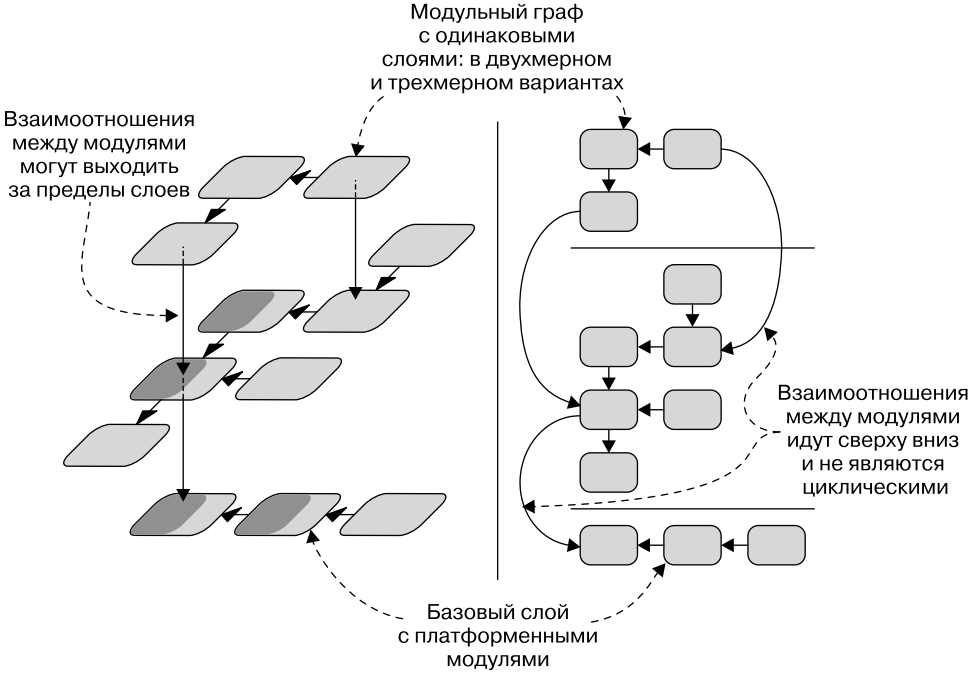


Рис. 12.2. Слои, диаграммы модулей могут быть сложены, добавляя третье измерение в вашу ментальную модель приложения. Поскольку они не разделяют загрузчики классов, слои хорошо изолированы друг от друга. (Как и любой хороший граф, этот может выглядеть перевернутым. Родительские слои находятся ниже своих дочерних элементов, поскольку слой с модулями платформы находится внизу)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Для *ServiceMonitor* это означает, что динамическая загрузка новой реализации наблюдателя требует создания нового слоя. Прежде чем переходить к этому в подразделе 12.4.3, подробнее рассмотрим существующие уровни и то, как их анализировать.

Все ли модули содержатся в слое? Почти. Как вы видели, технически говоря, безымянные модули не содержатся. Кроме того, существуют так называемые *динамические модули*, которые не обязательно должны принадлежать слою, но я не буду описывать их в данной книге. Помимо этих исключений, все модули являются частью слоя.

Загрузочный слой

Как насчет всех модулей приложений и платформенных модулей, которые были помещены в граф в этой книге? Они тоже должны принадлежать слою, верно?

Определение: загрузочный слой

Действительно, так и есть. При запуске JVM создает начальный уровень — загрузочный, содержащий модули приложения и платформы, которые были разрешены на основе параметров командной строки.

Загрузочный слой не имеет родителя и содержит три загрузчика классов:

- *загрузчик классов запуска* предоставляет всем классам, которые он загружает, все разрешения безопасности, поэтому команда JDK прилагает усилия для минимизации модулей, за которые отвечает; это несколько основных платформенных модулей, и главный из них — *java.base*;
- *загрузчик платформенных классов* загружает классы из всех других платформенных модулей; к нему можно получить доступ с помощью статического метода `ClassLoader::getPlatformClassLoader`;
- *загрузчик классов системы* или *приложения* загружает все классы из модуля и пути к классам, то есть отвечает за все модули приложения; доступ к нему можно получить с помощью статического метода `ClassLoader::getSystemClassLoader`.

Только загрузчик классов системы имеет доступ к пути к классам, вследствие чего из всех перечисленных лишь его безымянный модуль будет непустым. Следовательно, когда в разделе 8.2 говорится о безымянном модуле, всегда речь идет о загрузчике классов системы.

Как вы можете видеть на рис. 12.3, загрузчики классов не являются островками: у каждого загрузчика есть родительский элемент, и большинство реализаций, включая только что упомянутые три, сначала просят родительский класс загрузить класс, прежде чем пытаться найти его самостоятельно. Для трех перечисленных загрузчиков лишь загрузчик классов запуска является предком без родителей; загрузчик платформенных классов делегирует управление загрузчику классов запуска, а загрузчик классов системы передает управление загрузчику платформенных классов. Как следствие, загрузчик классов системы имеет доступ ко всем классам

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

приложений и JDK из загрузчиков классов запуска и загрузчиков платформенных классов.

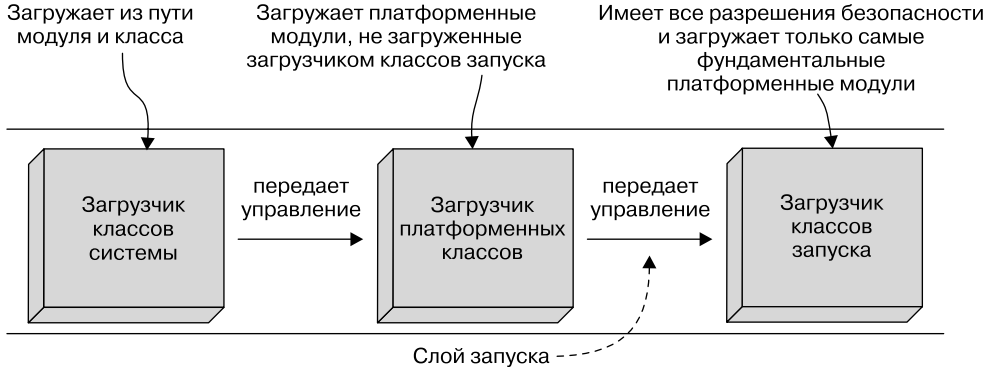


Рис. 12.3. Делегирование между тремя загрузчиками классов на слое запуска

12.4.2. Анализ слоев

Определение: ModuleLayer

Во время выполнения *слои представлены* экземплярами `java.lang.ModuleLayer`. Они могут быть запрошены для трех объектов, из которых состоит слой.

- Модули:
  - метод `modules()` возвращает модули, содержащиеся в слое, как `Set<Module>`;
  - метод `findModule(String)` ищет сам слой и все его слои-предки для модуля с указанным именем. Возвращает `Optional<Module>`, поскольку может не найти его.
- Родители слоя возвращаются как `List<ModuleLayer>` методом `parent()`.
- Загрузчик классов каждого модуля можно определить, вызвав `findLoader(String)` с именем модуля.

Затем есть метод `configuration`, который возвращает экземпляр `Configuration`, — подробнее об этом см. в подразделе 12.4.3.

Чтобы получить экземпляр `ModuleLayer`, можно запросить у любого модуля слой, к которому он принадлежит:

```
Class<?> type = // ... любой класс
ModuleLayer layer = type
    .getModule()
    .getLayer();
```

Последняя строка возвращает `null`, если тип получен из безымянного модуля или динамического модуля, который не принадлежит слою. При необходимости получить доступ к загрузочному слою можно вызвать статический метод `ModuleLayer::boot`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Итак, что можно узнать из экземпляров `ModuleLayer`? Несомненно, наиболее интересны методы `modules()` и `findModule(String)`, поскольку вместе с методами класса `Module` (см. подраздел 12.3.3) позволяют выполнять обход и анализ модульного графа.

Описание слоя модуля

Учитывая метод `describe(Module)` в листинге 12.7, можно описать целый уровень следующим образом:

```
private static String describe(ModuleLayer layer) {
    return layer
        .modules().stream()
        .map(ThisClass::describe)
        .collect(joining("\n\n"));
}
```

Поиск модулей в слоях и между ними

Можно также установить наличие или отсутствие определенных модулей, что может пригодиться, если зависимость от них необязательна (с помощью `requires static` (см. раздел 11.2)). В подразделе 11.2.4 я заявил, что было бы легко реализовать метод `isModulePresent(String)` с этой целью. Далее следует применение на практике того, что вы узнали о слоях, так что сделаем это шаг за шагом.

Начало кажется довольно тривиальным:

```
public boolean isModulePresent(String moduleName) {
    return ModuleLayer
        .boot()
        .findModule(moduleName)
        .isPresent();
}
```

Но это лишь показывает, присутствует ли модуль в загрузочном слое. Что, если дополнительные слои были созданы, а модуль находится в другом слое? Можно заменить загрузочный слой слоем, который содержит `isModulePresent`:

```
public  boolean isModulePresent(String moduleName) {
    return searchRootModuleLayer()
        .findModule(moduleName)
        .isPresent();
}

private ModuleLayer searchRootModuleLayer() {
    return this
        .getClass()
        .getModule()
        .getLayer();
}
```

Таким образом, `isModulePresent` ищет слой, содержащий его самого — назовем его `search`, — и все родительские слои. Но даже этого недостаточно. Модуль, вызывающий данный метод, может находиться на другом уровне с именем `call`, в котором `search`

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

является предком. (Запутались? См. рис. 12.4.) Тогда `search` не может посмотреть внутрь `call` и, следовательно, искать во всех возможных модулях. Нет, нам нужен модуль *вызывающей стороны*, чтобы использовать его слой в качестве корня для поиска.

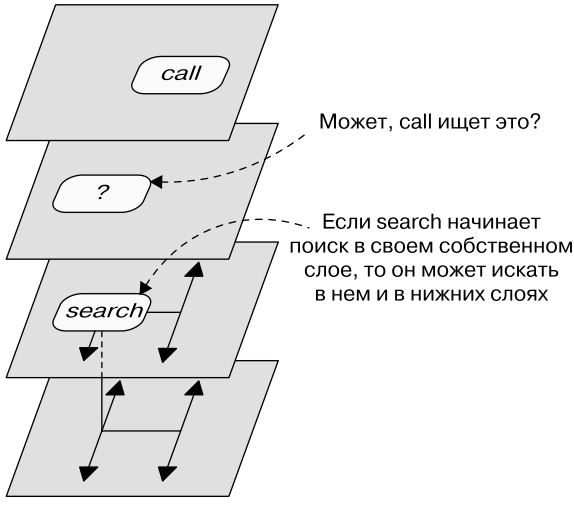


Рис. 12.4. Слой, попросивший найти модуль, сканирует только себя и своих родителей (на этом графике вниз). Таким образом, если search запрашивает собственный слой, то может игнорировать слои, которые call, модуль, инициировавший поиск, может видеть, тем самым рискуя вернуть неправильный результат. Вот почему важно запросить слой call

В листинге 12.9 реализован метод `getCallerClass`, который определяет класс вызывающей стороны с помощью API обхода стека, представленного в Java 9.

Листинг 12.9. Новый API для обхода стека вызовов

```
private Class<?> getCallerClass() {
    return StackWalker
        .getInstance(RETAIN_CLASS_REFERENCE)
        .walk(stack -> stack
        .filter(frame ->
            frame.getDeclaringClass() != this.getClass())
        .findFirst()
        .map(StackFrame::getDeclaringClass)
        .orElseThrow(IllegalStateException::new)
    );
}
```

Статический фабричный метод для получения экземпляра StackWalker, где каждый фрейм имеет ссылку на объявленный класс

StackWalker::walk ожидает функцию от Stream<StackFrame> для произвольного объекта. Он создает ленивый вид стека и сразу вызывает функцию с ним в качестве параметра. Объект, который возвращает функция, затем возвращается функцией walk

Получает данный класс

Было бы странно, если бы такого фрейма не существовало...

Вы, должно быть, заинтересованы в первом фрейме из класса, который не принадлежит данному (это должен быть вызывающий объект!); теперь у вас есть дополнительный <StackFrame>

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Имея это в наборе инструментов, вы держите модуль вызывающего объекта под рукой:

```
public boolean isModulePresent(String moduleName) {
    return searchRootModuleLayer()
        .findModule(moduleName)
        .isPresent();
}

private ModuleLayer searchRootModuleLayer() {
    return getCallerClass()
        .getModule()
        .getLayer();
}
```

Вот и все, что касается анализа слоев. Теперь мы можем наконец перейти к самой захватывающей части: загрузить новый код в работающее приложение, создав новые слои.

12.4.3. Создание модульных слоев

Только часть приложений, написанных на Java, должна динамически загружать код во время выполнения. Вместе с тем такие приложения на практике оказываются более важными. Возможно, самое известное — Eclipse, который уделяет большое внимание плагинам, но и серверы приложений, такие как WildFly и GlassFish, должны загружать код из одного или нескольких приложений одновременно. Как обсуждается в подразделе 15.3.3, OSGi тоже может динамически загружать и выгружать связки (так он называет модули).

Все они имеют одинаковые основные требования к механизму, который используют для загрузки плагинов, приложений, пакетов и других новых фрагментов работающей JVM:

- должен давать возможность раскручивать фрагмент из набора JAR во время выполнения;
- должен позволять взаимодействие с загруженными фрагментами;
- должен позволить изоляцию между различными фрагментами.

До системы модулей это делалось с помощью загрузчиков классов. Если коротко, то для новых JAR-файлов был создан новый загрузчик классов. Он делегирован другому загрузчику классов, такому как системный загрузчик классов, который предоставил ему доступ к другим классам в работающей JVM. Каждый класс, идентифицируемый его полным именем, может существовать только один раз для одного загрузчика классов, однако его легко можно загрузить несколькими загрузчиками. Это изолирует фрагменты и дает каждому возможность придумывать собственные зависимости, не конфликтуя с другими фрагментами.

Система модулей никоим образом не изменила это. Сохранение иерархий загрузчиков классов было одной из основных причин реализации системы модулей ниже загрузчиков классов. Было добавлено только понятие слоев вокруг

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

загрузчиков классов, которые обеспечивают интеграцию с модулями, загруженными во время запуска. Посмотрим, как можно его создать. (Существует также вариант *ServiceMonitor*, создающий слои в ветке `feature-layers`.)

Создание конфигурации

Важным ингредиентом `ModuleLayer` является `Configuration`. Создание `Configuration` запускает процесс разрешения модуля (см. подраздел 3.4.1), и полученный экземпляр представляет успешно разрешенный модульный. Самая простая форма для создания конфигурации — это статические фабричные методы `resolve` и `resolveAndBind`. Единственное различие между ними состоит в том, что второй связывает сервисы, тогда как первый — нет.

И `resolve`, и `resolveAndBind` принимают четыре одинаковых аргумента:

- `ModuleFinder before` запрашивается для поиска модулей перед просмотром родительских конфигураций;
- `List<Configuration> parent` — это конфигурации родительских слоев;
- `ModuleFinder after` просит найти модули после просмотра родительских конфигураций;
- `Collection<String> roots` — это корневые модули для процесса разрешения.

Создать `ModuleFinder` для пути модуля так же просто, как вызвать `ModuleFinder.of(Path...)`. Распространены попытки ссылаться на максимальное количество модулей из родительского слоя, так что средство поиска `before` часто создается без аргумента и, следовательно, не может найти никаких модулей.

В общем случае, когда требуется создать конфигурацию с одним родителем, проще вызвать методы экземпляра `resolve` и `resolveAndBind`. Они не имеют родительского аргумента `List<Configuration>` и используют текущую конфигурацию в качестве родительской.

Допустим, вы хотите создать конфигурацию с загрузочным уровнем в качестве родительского, эмулирующего команду запуска `java --module-path mods --module root`, но без привязки сервиса. Для этого вы можете вызвать `resolve` (чтобы сервисы не были связаны) в конфигурации загрузочного уровня (сделать его родительским) и передать модуль поиска, который просматривает каталог `mods`. Листинг 12.10 показывает это: в нем создается конфигурация, эмулирующая `java --module-path mods --module initial` минус привязка сервиса.

В качестве второго примера вернемся к сценарию, в котором вы хотите, чтобы *ServiceMonitor* начинал наблюдать за новыми сервисами во время выполнения, для чего необходимо загрузить новые реализации `ServiceObserver`. Первый шаг — создание конфигурации с текущим слоем в качестве родительского, который ищет модули по указанному пути.

Вы используете сервисную инфраструктуру системы модулей для своих сервисов, поэтому вызываете `resolveAndBind`. Вы можете полагаться исключительно на данный механизм, чтобы найти все необходимые модули (и их зависимости),

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

вследствие чего даже не нужно указывать корневые модули. В листинге 12.11 представлена реализация.

Листинг 12.10. Эмуляция java --module-path mods --module

Не нужно искать модули, прежде чем заглядывать в родительский граф

Средство поиска модулей, которых нет в родительском графе, просматривает каталог mods

```
ModuleFinder emptyBefore = ModuleFinder.of();
ModuleFinder modulePath = ModuleFinder.of(Paths.get("mods"));
Configuration bootGraph = ModuleLayer.boot().configuration();
Configuration graph = bootGraph #C
         .resolve(emptyBefore, modulePath, List.of("initial"));
```

Определяет конфигурацию загрузочного уровня как родительскую, вызывая на нем функцию resolve (resolveAndBind будет связывать сервисы)

Листинг 12.11. Конфигурация, которая связывает все модули из указанных путей

```
private static Configuration createConfiguration(Path[] modulePaths) {
    return getThisLayer()
        .configuration()
        .resolveAndBind(
            ModuleFinder.of(),
            ModuleFinder.of(modulePaths),
            Collections.emptyList()
        );
}
```

Возвращает слой, к которому принадлежит класс, содержащий createConfiguration

Вызывается так, чтобы сервисы были разрешены

Вы полагаетесь на привязку сервисов, которые могли выполнить работу за вас, и извлекаете нужные модули, поэтому не определяете корневые модули

Создание ModuleLayer

Как описано в подразделе 12.4.1, слой состоит из модульного графа, загрузчиков классов и ссылок на родительские уровни. Простая форма для создания модуля доступна с помощью статического метода `defineModules(Configuration, List<ModuleLayer>, Function<String, ClassLoader>)`:

- вы уже знаете, как получить экземпляры `Configuration`;
- `List<ModuleLayer>` — это родители;
- `Function<String, ClassLoader>` сопоставляет имя каждого модуля с загрузчиком классов, который нужно назначить ответственным за данный модуль.

Метод возвращает `Controller`, который пригоден для дальнейшего редактирования графа модуля путем добавления ребер чтения или экспорта/открытия пакетов перед вызовом `layer()`, возвращающим `ModuleLayer`.

Есть несколько альтернативных методов, которые можно вызвать, основываясь на `defineModules`.

- `DefineModulesWithOneLoader` использует один загрузчик классов для всех модулей. Загрузчик класса, указанный в качестве аргумента метода, становится его родителем.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- `DefineModulesWithManyLoaders` задействует отдельный загрузчик классов для каждого модуля. Загрузчик класса, указанный в качестве аргумента метода, становится родителем каждого из них.
- Существуют варианты каждого метода, который можно вызывать в экземпляре `ModuleLayer` и использовать этот экземпляр в качестве родительского уровня; они возвращают созданный слой вместо промежуточного `Controller`.

В продолжение квеста по динамической загрузке реализаций `ServiceObserver` следующим шагом будет создание фактического слоя из конфигурации. Это довольно просто, как показывает листинг 12.12.

Листинг 12.12. Создание слоя из конфигурации

```
Создает конфигурацию, как в листинге 12.11
                                  getThisLoader возвращает загрузчик класса,
                                  который загружает класс, содержащий createLayer
  private static ModuleLayer createLayer(Path[] modulePaths) {
      Configuration configuration = createConfiguration(modulePaths);
      ClassLoader thisLoader = getThisLoader();
      return getThisLayer()
          .defineModulesWithOneLoader(configuration, thisLoader);
  }
То же самое с getThisLayer в листинге 12.11
                                  Вам нужен только один загрузчик для всех модулей
                                  с данным слоем в качестве родительского, поэтому
                                  стоит вызвать для него defineModulesWithOneLoader
```

Последний шаг — проверить, содержит ли недавно созданный слой `ServiceObserver`, который может обслуживать сервис, подлежащий вашему наблюдению. Для этого можно использовать перегрузку `ServiceLoader::load`, ожидающую `ModuleLayer` в дополнение к типу сервиса, который он ищет. Семантика должна быть ясной: изучите этот уровень (и его предков) при поиске поставщиков (листинг 12.13).

Листинг 12.13. Открытие поставщиков услуг на новом уровне (и их предков)

```
  private static void registerNewService(
          String serviceName, Path... modulePaths) {
      ModuleLayer layer = createLayer(modulePaths);
      Stream<ServiceObserverFactory> observerFactories = ServiceLoader
          .load(layer, ServiceObserverFactory.class).stream()
          .map(Provider::get);
      Optional<ServiceObserver> observer = observerFactories
          .map(factory -> factory
              .createIfMatchingService(serviceName))
          .flatMap(Optional::stream)
          .findFirst();
      observer.ifPresent(monitor::addServiceObserver);
  }
```

Создает слой, как в листинге 12.12

Остальное — обычное дело сервиса — найти observer для имени сервиса

Использует вариант ServiceLoader::load, который принимает новый слой

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Если этого было недостаточно, то имеется еще несколько моментов, которые мы едва затронули, из того, какие еще действия доступны со слоями модулей:

- создание конфигураций и слоев с несколькими родителями или несколькими загрузчиками классов;
- использование слоев для загрузки нескольких версий одного и того же модуля;
- изменение модульного графа с помощью `Controller`, например, чтобы экспортировать или открыть модули, прежде чем превратить их в модуль `ModuleLayer`;
- непосредственная загрузка специфичных классов из созданных слоев в качестве точек входа во фрагмент, в отличие от использования сервисов JPMS.

Подробнее узнать об этом можно из превосходного Javadoc о задействованных методах, особенно в `ModuleLayer` и `Configuration`. Или перейдите к разделу 13.3, в котором хорошо продемонстрированы некоторые из этих возможностей.

Резюме

- Модули, которые рефлексированы в коде.
  - В большинстве случаев директивы `exports` не подходят для того, чтобы сделать классы доступными для рефлексии, поскольку классы, разработанные для основанных на рефлексии сред, редко могут быть частью открытого API модуля. При квалифицированном экспорте вам, возможно, понадобится соединить модуль с реализацией вместо стандарта. И экспорт не поддерживает глубокую рефлексию непривилегированных областей и методов.
  - По умолчанию не стоит использовать `exports` — лучше задействовать директивы `opens`, чтобы открывать пакеты для рефлексии.
  - Директива `opens` имеет тот же синтаксис, что и `exports`, но работает иначе: открытый пакет недоступен во время компиляции; и все типы и члены, включая непубличные, в открытом пакете доступны во время выполнения. Эти свойства тесно связаны с требованиями фреймворков, основанных на рефлексии, что делает директивы `opens` выбором по умолчанию при подготовке модулей к рефлексии.
  - Квалифицированный вариант `opens ... to` открывает пакет только для именованных модулей. Поскольку обычно весьма очевидно, какие пакеты рефлексируют те или иные фреймворки, сомнительно, что квалификация директив `open` придает большую ценность.
  - Если рефлексирующий фреймворк разделен на стандарт и его реализации (как в JPA и Hibernate, EclipseLink и т. д.), то технически возможно открыть только пакет для стандарта, который затем может использовать API рефлексии, чтобы открыть его для конкретной реализации. Это пока реализовано

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

не слишком широко, хотя в настоящее время квалифицированным открытиям необходимо назвать конкретные модули реализации.

- Параметр командной строки `--add-opens` имеет тот же синтаксис, что и `--add-exports`, и работает как квалифицированные открытия. Открытие платформенных модулей из командной строки для доступа к их внутренним элементам — обычное явление при переходе на Java 9+, но можно использовать его и для разбиения приложения на другие модули в случае абсолютной необходимости в этом.
- Если начать объявление модуля с `open module` (а не просто с `module`), то открываются все пакеты в данном модуле. Это хорошее решение, когда модуль содержит много пакетов, которые необходимо открыть, но следует тщательно оценить, действительно ли это нужно или можно исправить. В идеале открытые модули в основном используются во время модуляризации, предваряя перевод модуля в более чистое состояние, в котором меньше внутренних компонентов.

❑ Код, рефлексирующий модули.

- Рефлексия связана с теми же правилами доступности, что и обычный код. Относительно необходимости чтения модуля, к которому вы обращаетесь, API рефлексии упрощает задачу, неявно добавляя ребро чтения. Применительно к экспортированным или открытым пакетам автор рефлексирующего кода ничего не может с этим поделать, если владелец модуля не подготовил для него свой модуль. (Единственное решение — параметр командной строки `--add-opens`.)
- Данное обстоятельство делает все более необходимым информирование пользователей о строгой инкапсуляции и о том, к каким пакетам нужен доступ вашему модулю. Хорошо документируйте проект и сделайте источник легкодоступным.
- Обязательно правильно обрабатывайте исключения, которые вызываются из-за строгой инкапсуляции, чтобы можно было предоставить пользователям информативное сообщение об ошибке, возможно, со ссылкой на документацию.
- Попробуйте использовать обработчики переменных вместо API рефлексии. Они обеспечивают бóльшую безопасность типов, более производительны и позволяют выразить потребность в доступе в вашем API начальной загрузки, запрашивая экземпляры `Lookup`.
- Экземпляр `Lookup` предлагает всем, кто применяет его, такую же доступность, как и создавший его модуль. Поэтому, когда ваши пользователи создают экземпляр `Lookup` в своем модуле и передают его в ваш фреймворк, можно получить доступ к их внутренним компонентам.
- Новые классы `Module` и `ModuleDescriptor` являются частью API рефлексии и предоставляют доступ ко всей информации, касающейся модуля, такой как его имя, зависимости и экспортированные или открытые пакеты. Подходят для анализа фактического графа модуля во время выполнения.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- Используя этот API, модули могут также изменять собственные свойства и экспортировать или открывать пакеты либо добавлять ребра чтения для других модулей. Как правило, невозможно изменить другие модули, за исключением того, что каждый модуль, для которого был открыт пакет другого модуля, может открыть данный пакет для третьего модуля.

- Код, который динамически загружает модули.
  - Загрузчики классов позволяют динамически загружать код в работающую программу. Эта концепция не меняется в системе модулей, но предоставляет модульную оболочку для загрузчиков классов со слоями. Слой инкапсулирует загрузчик классов и граф модулей, и при создании последнего загруженные модули подвергаются всем проверкам согласованности и правил доступности, которые предлагает система модулей. Следовательно, слои можно использовать с целью обеспечить надежную конфигурацию и строгую инкапсуляцию для загруженных модулей.
  - При запуске JVM создает загрузочный уровень, состоящий из трех загрузчиков классов и всех платформенных модулей и приложений, которые были первоначально разрешены. Доступ к нему можно получить с помощью статического метода `ModuleLayer::boot`, а возвращенный экземпляр `ModuleLayer` — использовать для анализа всего графа модуля.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

13 Версии и модули: возможное и невозможное

В этой главе

- Почему система модулей не работает с информацией о версии.
- Запись информации о версии.
- Анализ информации о версии во время выполнения.
- Загрузка нескольких версий модуля.

Как кратко упомянуто в подразделе 1.5.6, JPMS не поддерживает версии модулей. Но тогда для чего нужен `jar --module-version`? И разве подраздел 12.3.3 не показал, что `ModuleDescriptor` может по крайней мере сообщать версию модуля? Эта глава проясняет ситуацию и рассматривает версии модулей с разных точек зрения.

Сначала мы обсудим, каким образом система модулей может поддерживать версии и почему она этого не делает (см. раздел 13.1). Однако она позволяет записывать и оценивать информацию о версии, и мы рассмотрим это далее (см. раздел 13.2). Последним в списке является святой Грааль: запуск разных версий одного и того же модуля (см. раздел 13.3). Несмотря на отсутствие для его встроенной поддержки, существуют способы выполнить запуск, приложив некоторые усилия.

К концу этой главы вы получите четкое представление об ограниченной поддержке версий системой модулей. Данная информация поможет проанализировать ваше приложение и даже вполне пригодна, чтобы предупредить о потенциальных

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

проблемах. Возможно, что более важно, вы также будете знать причины ограничений и то, можно ли ожидать их изменения. Кроме того, вы узнаете способы запуска нескольких версий одного и того же модуля, как и то, что это редко будет стоить затраченных усилий.

13.1. Отсутствие поддержки версий в JPMS

Java 8 и более ранние версии не имеют понятия о версиях. Как описано в подразделе 1.3.3, это может привести к непредвиденному поведению во время выполнения, когда потенциально единственное решение — выбор версий зависимостей, отличных от тех, которые вы хотели бы использовать. Это прискорбно, и когда система модулей была впервые задумана, одной из ее целей числилось исправить эту ситуацию.

Однако этого не произошло. Система модулей, которая сейчас работает на Java, все еще сравнительно слепа к версиям. Она ограничена записью версии модуля или зависимости (см. раздел 13.2).

Но почему так? Разве система модулей не может поддерживать несколько версий одного и того же модуля (см. подраздел 13.1.1)? Если нет, то не может ли она хотя бы предоставить набор модулей и требований к версии в качестве входных данных и выбрать одну версию для каждого модуля (см. подраздел 13.1.2)? Ответ на оба вопроса — нет, и я хочу объяснить почему.

13.1.1. Нет поддержки нескольких версий

Казалось бы, простое решение конфликтов версий — разрешить запуск двух версий одного и того же JAR. Непосредственно. Так почему же система модулей не может просто сделать это? Чтобы ответить на данный вопрос, вы должны знать, как Java загружает классы.

Как загрузка классов предотвращает несколько версий

Как обсуждалось при рассмотрении затенения в подразделе 1.3.2, JVM — или, точнее, ее загрузчики классов — идентифицирует классы по их полностью определенному имени, например `java.util.List` или `monitor.observer.ServiceObserver`. Чтобы загрузить класс из пути класса, загрузчик класса приложения сканирует все JAR-файлы, пока не обнаружит класс с конкретным искомым именем, который затем загружается.

ВАЖНАЯ ИНФОРМАЦИЯ

Критическое наблюдение заключается в том, что не имеет значения, содержит ли другой JAR в пути к классам класс с точно таким же именем, — он никогда не будет загружен. Другими словами, загрузчик классов работает с предположением, что каждый класс, идентифицируемый его полностью определенным именем, существует ровно один раз.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Возвращаясь к нашему желанию запустить несколько версий одного и того же модуля, препятствие очевидно: такие модули обязательно должны содержать классы с одинаковым полностью определенным именем, и без каких-либо изменений JVM увидит только одну версию из всего списка. Как выглядят эти изменения?

Изменения в загрузке классов, которые позволят использовать несколько версий

Первый вариант, позволяющий применять несколько классов с одинаковыми именами, — переписать весь механизм загрузки классов, чтобы отдельный загрузчик классов мог обработать данный случай. Это было бы огромной инженерной задачей, поскольку предположение, что каждый загрузчик классов имеет не более одного класса с любым именем, пронизывает всю JVM. Помимо огромных усилий, это несет еще и большой риск: данное изменение будет инвазивным и, следовательно, почти гарантированно несовместимым в обратном направлении.

Второй вариант — разрешить нескольким классам с одним и тем же именем делать, например, OSGi: использовать отдельный загрузчик классов для каждого модуля (рис. 13.1). Это было бы относительно просто, но также могло бы вызвать проблемы совместимости.

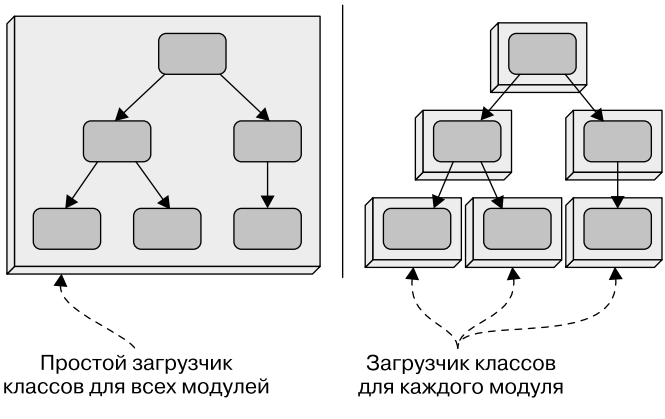


Рис. 13.1. JPMS использует один и тот же загрузчик классов для всех модулей приложения *(слева)*, но вполне возможно, что вместо него он может применить отдельный загрузчик для каждого модуля *(справа)*. Однако во многих случаях это чревато изменением поведения приложения

Один из потенциальных источников проблем таков: некоторые инструменты, платформы и даже приложения делают конкретные предположения о точной иерархии загрузчиков классов. (По умолчанию есть три загрузчика классов, ссылающиеся друг на друга, — это не изменилось в Java 9. Подробности объясняются в описании загрузочного слоя в подразделе 12.4.1.) Если поместить каждый модуль в собственный загрузчик классов, то это значительно изменит данную иерархию и, вероятно, разрушит большинство проектов.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В изменении иерархии скрыта еще одна хитрая деталь. Даже если вы захотите требовать, чтобы проекты адаптировались к этому изменению для запуска из пути модуля, что произойдет, если они будут выполняться из пути класса? Получат ли JAR из пути к классам отдельный загрузчик классов?

- Если да, то проекты, у которых есть проблемы с измененной иерархией загрузчика классов, не будут работать как модули, а также не станут запускаться в Java 9+.
- Если нет, то им необходимо различать две разные иерархии загрузки классов и правильно взаимодействовать с каждой из них в зависимости от того, на каком пути они приземлились.

Ни одно из этих воздействий на совместимость или пути миграции не является приемлемым в случае применения ко всей экосистеме.

ПРИМЕЧАНИЕ

Вес этих проблем различен для OSGi. Он предлагает функции, без которых большинство использующих его приложений не могут обойтись, поэтому можно ожидать, что их разработчики приложат больше усилий. Java 9+, с другой стороны, также должен работать для проектов без системы модулей. OSGi лишь подписчик, вследствие чего какую-либо версию, не работающую для какого-либо конкретного проекта, можно игнорировать. Очевидно, что случай с Java 9+ здесь не подходит.

ВАЖНАЯ ИНФОРМАЦИЯ

Другая причина, по которой конкретный загрузчик классов для каждого JAR может быть проблематичным, связана с равенством классов. Предположим, один и тот же класс был загружен двумя разными загрузчиками классов. Их экземпляры Class<?> не равны, поскольку загрузчик классов всегда включен в данную проверку. Так? Кого это вообще волнует, верно?

Итак, если у вас есть экземпляр каждого класса и вы сравниваете оба, то что произойдет в первую очередь при сравнении на `equals`? Это будет либо `this.getClass() == other.getClass()`, либо проверка `instanceof`. В таком случае результат всегда будет ложным, поскольку два класса не равны.

Это значит, что в двух версиях Guava, например, `mutimap1.equals(multimap2)` всегда будет ложным, независимо от того, какие элементы содержатся в двух экземплярах `Multimap`. Кроме того, нельзя преобразовать экземпляр класса из одного загрузчика классов в тот же класс, загруженный из другого, поэтому `(Multimap) multimap2` может завершиться ошибкой:

```
static boolean equalsImpl(
        Multimap<?, ?> multimap,
        @NullableDecl Object object) {
    if (object == multimap) {
        return true;
    }
```

Экземпляр Multimap, в котором вызывается equals. Метод выполняется в контексте загрузчика классов

Объект, переданный в вызов equals. Предполагается, что это экземпляр Multimap из другого загрузчика классов

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
    if (object instanceof Multimap) {
        Multimap<?, ?> that = (Multimap<?, ?>) object;
        return multimap.asMap().equals(that.asMap());
    }
    return false;
}
```

Объект имеет тип Multimap, но он из другого загрузчика классов, поэтому проверка instanceof всегда завершается неудачей

Было бы неплохо узнать, сколько проектов будет сломано именно из-за данной детали. Не существует никакого способа узнать, но я думаю, что много. По сравнению с этим главы 6 и 7 совершенно безобидны.

ПРИМЕЧАНИЕ

Кстати, все, что мы сейчас обсудили, относится и к разделенным пакетам (см. раздел 7.2). Разве не было бы хорошо, будь системе модулей безразлично, содержат ли два модуля один и тот же пакет и могут ли быть отдельными? Так и было бы, но мы столкнулись бы с теми же проблемами, которые исследовали только что.

Определенное нами до сих пор означает лишь то, что система модулей не допускает несколько версий одного и того же модуля «из коробки». Отсутствие встроенной поддержки вовсе не значит, что она категорически невозможна. Взгляните на раздел 13.3, чтобы узнать, как заставить это работать.

13.1.2. Нет поддержки выбора версии

Если система модулей не может загрузить несколько версий одного и того же модуля, то почему она не может хотя бы выбрать для нас правильные версии? Это, конечно, теоретически вероятно, но, к сожалению, так нельзя — позвольте мне объяснить почему.

Как инструменты сборки работают с версиями

Инструменты сборки, такие как Maven и Gradle, постоянно работают с версионными JAR-файлами. Для каждого JAR они знают его собственную версию и версию его зависимостей. Учитывая плечи гигантов, на которых стоит столько проектов, вполне естественно, что они хранят деревья глубоких зависимостей, содержащие одни и те же файлы JAR несколько раз, возможно, с разными версиями.

Хотя приятно знать, сколько разных версий требует JAR, это не меняет того факта, что не все попадают в путь классов. В противном случае вы столкнетесь с такими проблемами, как затенение (см. подраздел 1.3.2) и прямые конфликты версий (см. подраздел 1.3.3), которые будут угрожать стабильности вашего проекта.

ВАЖНАЯ ИНФОРМАЦИЯ

Когда приходит время компилировать, тестировать или запускать проект, инструменты сборки должны сгладить это дерево в список, содержащий

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

каждый JAR-файл только один раз (рис. 13.2). По сути, они должны выбрать версию для каждого артефакта. Это нетривиальный процесс, особенно если артефакты могут определять диапазон приемлемых версий для каждой зависимости. Поскольку процесс нетривиален, он также не особенно прозрачен. Может быть трудно предсказать, какие версии выберут Maven или Gradle, и неудивительно, что они не обязательно выбирают одни и те же при одинаковых обстоятельствах.

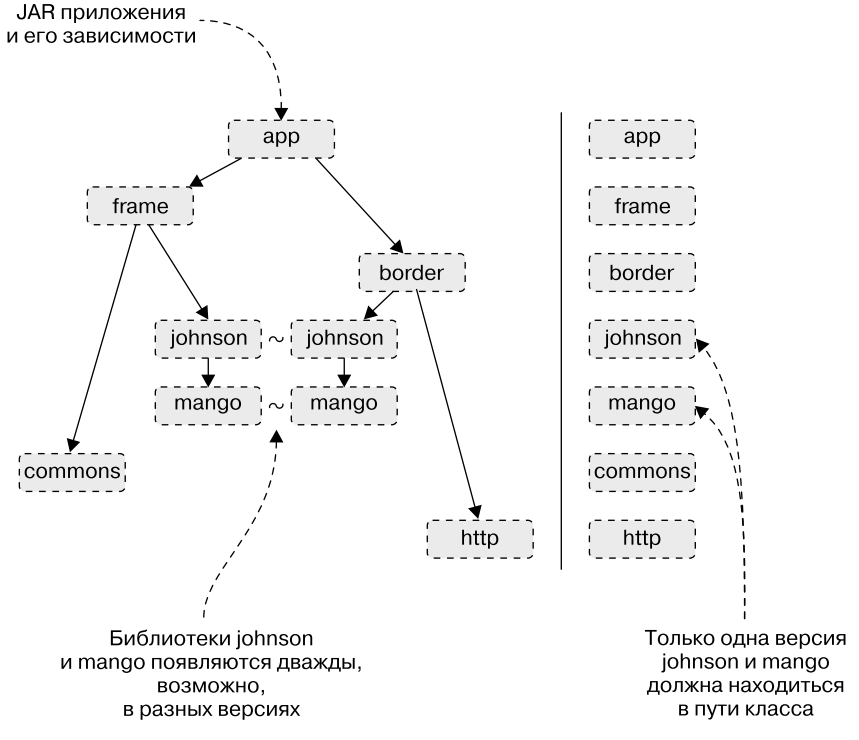


Рис. 13.2. Дерево зависимостей приложения *(слева)* может содержать один и тот же JAR несколько раз, например johnson и mango, потенциально в разных версиях. Чтобы работать с путем класса, это дерево следует сократить до набора, который содержит каждый JAR только один раз *(справа)*

Почему система модулей не выбирает версии

Теперь оставим инструменты сборки и поговорим о системе модулей. Как вы увидите в разделе 13.2, модули могут записывать собственные версии и их зависимости. Предположим, что система не может запустить несколько экземпляров одного и того же модуля, не может ли она тогда выбрать одну версию каждого?

Доиграем до конца. В этом гипотетическом сценарии JPMS будет принимать несколько версий одного и того же модуля в пути модуля. При построении графа для каждого модуля JPMS будет решать, какую версию выбрать.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

Иначе говоря, JPMS теперь будет дублировать инструменты сборки. И поскольку они по-разному решают задачу, система модулей будет вести себя несколько иначе, чем большинство (вероятно, все) из них. Что еще хуже, вследствие того, что Java основан на стандарте, точное поведение, вероятно, придется стандартизировать, а это усложнит его развитие с течением времени.

Вдобавок ко всему этому необходимо приложить усилия по реализации и сопровождению алгоритма выбора версии. Последний гвоздь в крышке гроба — производительность: если компилятору и JVM придется запустить данный алгоритм, прежде чем они смогут начать реальную работу, то это значительно увеличит время компиляции и запуска. Как видите, выбор версии — недешевая функция, и есть смысл в том, что Java ее не использует.

13.1.3. Что может принести будущее

Итак, система модулей не зависит от версии, то есть информация о версии не влияет на ее поведение. Таково положение дел на сегодняшний день. Многие разработчики надеются, что Java в будущем поддержит какую-либо из этих функций. Если вы один из них, то я не хочу вас обламывать. Насколько бы ни была мала вероятность такого развития, нет гарантий того, что ситуация не изменится. Однако я на такие чудеса не рассчитываю.

ВАЖНАЯ ИНФОРМАЦИЯ

Марк Рейнхольд, главный архитектор Java Platform Group в Oracle и ведущий специалист по спецификации системы модулей, неоднократно и публично заявлял, что не видит поддержки версий в будущем Java. Учитывая значительные инвестиции, которые потребуются для такой функции, и ее сомнительную отдачу, я могу понять, как он пришел к подобному решению.

То есть мы все еще должны бороться с проблемами версий. Может быть, это говорит мой стокгольмский синдром, но я не считаю эту борьбу напрасной. Работа, иногда тяжелая, унифицирующая диапазоны версий по проекту и обеспечивающая набор уникальных JAR-файлов, которые поддерживают приложение, на самом деле дает свои преимущества.

Допустим, у вас нет стимула сделать это. Сколько еще JAR-файлов проект будет перетаскивать на путь класса или модуля? Насколько большим он станет и насколько усложнится отладка? Нет, я думаю, что позволить конфликтующим версиям работать «из коробки» было бы плохой идеей.

Тем не менее факт остается фактом: в некоторых случаях конфликт версий останавливает важную работу на своем пути или делает невозможным критические обновления без одновременных обновлений тонны других зависимостей. Для этого было бы неплохо иметь переключатель командной строки, такой как `java --one-class-loader-per-module`, который можно было бы попробовать в черный день. Увы, его еще нет (пока?).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

13.2. Запись информации о версии

Как мы рассмотрели в подробностях секунду назад, система модулей не обрабатывает информацию о версии. Интересно, что это ограничение все-таки *позволяет* записывать и получать доступ к данной информации. Поначалу это может показаться немного странным, но оказывается полезным при отладке приложения.

Сначала посмотрим, как записывать информацию о версии во время компиляции и упаковки (см. подраздел 13.2.1), а затем обсудим, где можно увидеть эту информацию и какие преимущества она предоставляет (см. подраздел 13.2.2). Запись и оценка информации о версии демонстрируется в ветке `feature-versions` приложения *ServiceMonitor*.

13.2.1. Запись версий при сборке модулей

Определение: --module-version

Команды `javac` и `jar` принимают параметр командной строки `--module-version ${`*`версия`*`}`. Они встраивают данную версию, которая может быть произвольной строкой, в дескриптор модуля.

Независимо от того, используется ли опция, если модуль скомпилирован с зависимостью, записавшей его версию, компилятор тоже добавит эту информацию в дескриптор модуля. Это значит, что дескриптор может содержать версию самого модуля, а также всех зависимостей, относительно которых тот был скомпилирован.

Команда `jar` переопределяет версию модуля при условии, что она присутствовала ранее. Таким образом, если `--module-version` используется как в `jar`, так и в `javac`, то имеет значение только значение, данное `jar`.

В листинге 2.5 показано, как скомпилировать и упаковать модуль *monitor*, но вам не нужно возвращаться. Обновление команды `jar` для записи версии тривиально:

```
$ jar --create
    --file mods/monitor.jar
    --module-version 1.0
    --main-class monitor.Monitor
    -C monitor/target/classes .
```

Видите, это так же просто, как вставить `--module-version 1.0`. Поскольку сценарий компилирует и сразу упаковывает модуль, нет необходимости добавлять его и в `javac`.

Увидеть, добились ли вы успеха, можно, запросив `jar --describe-module` (см. подраздел 4.5.2):

```
$ jar --describe-module --file mods/monitor.jar

> monitor@1.0 jar:.../monitor.jar/!module-info.class
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
> requires java.base mandated
> requires monitor.observer
# директивы requires урезаны
> contains monitor
> main-class monitor.Main
```

Версия находится прямо в первой строке: `monitor@1.0`. Почему версии зависимостей не отображаются? В этом конкретном случае я их не записывал, но *java.base* определенно имеет их и они тоже не появляются. Действительно, `--describe-module` не выводит эту информацию — ни `jar`-, ни `java`-вариант.

Для получения доступа к версиям зависимостей модуля нужен другой подход. Посмотрим, где появляется информация о версии и как можно получить к ней доступ.

13.2.2. Доступ к версиям модуля

Версии, записанные во время компиляции и упаковки, отображаются в разных местах. Как вы только что видели, и `jar --describe-module`, и `java --describe-module` выводят версию модуля.

Информация о версии в трассировке стека

Трассировки стека — тоже важные места. Если код выполняется в модуле, то имя последнего печатается для каждого фрейма стека вместе с именами пакетов, классов и методов. Хорошая новость в том, что информация о версии также включена:

```
> Exception in thread "main" java.lang.IllegalArgumentException
>     at monitor@1.0/monitor.Main.outputVersions(Main.java:46)
>     at monitor@1.0/monitor.Main.main(Main.java:24)
```

Ничего революционного, но определенно приятное дополнение. Допустим, ваш код ведет себя неправильно по, казалось бы, загадочным причинам. Это может быть вызвано проблемами с версиями. Выставив их на видное место, легче заметить, если они подозрительны.

ВАЖНАЯ ИНФОРМАЦИЯ

Я убежден, что информация о версии может оказать большую помощь. Я настоятельно рекомендую вам сконфигурировать инструмент сборки для ее записи.

Информация о версии модуля в API рефлексии

Возможно, самое интересное место для обработки информации о версии — это API рефлексии. (В дальнейшем вам нужно знать о `java.lang.ModuleDescriptor`. Ознакомьтесь с подразделом 12.3.3, если вы этого еще не сделали.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

Как вы могли видеть в листинге 12.7 и видите снова в листинге 13.1, класс ModuleDescriptor имеет метод rawVersion(). Он возвращает Optional<String>, который содержит строку версии в точности так, как она была передана в --module-version, или пустую, если опция не использовалась.

Кроме того, существует version(), которая возвращает Optional<Version>, где Version — это внутренний класс ModuleDescriptor, анализирующий необработанную версию в сопоставимом представлении. Если исходной версии нет или ее анализ не выполнен, то Optional пуст.

Листинг 13.1. Доступ к необработанной и проанализированной версии модуля

```
ModuleDescriptor descriptor = getClass()
    .getModule()
    .getDescriptor();
String raw = descriptor
    .rawVersion()
    .orElse("unknown version");
String parsed = descriptor
    .version()
    .map(Version::toString)
    .orElse("unknown or unparsable version");
```

Возвращает Optional<String>, который пуст, если --module-version не использовался

Возвращает Optional<Version>, который пуст, если rawVersion() имеет значение или если необработанная версия не может быть проанализирована

Информация о версии зависимости в API рефлексии

API рефлексии определяет собственную версию модуля. Вы все еще не видели, как получить доступ к версиям, которые были записаны для зависимостей. Или видели? Листинг 12.8, показывающий вывод практически всего, что может предложить `ModuleDescriptor`, содержит такой фрагмент:

```
[] module monitor.persistence @ [] {
    requires [
        hibernate.jpa,
        mandated java.base (@9.0.4),
        monitor.statistics]
    [...]
}
```

Видите `@9.0.4`? Это часть вывода `Requires::toString`. `Requires` — это еще один внутренний класс `ModuleDescriptor`, и он представляет директиву `requires` в дескрипторе модуля.

ВАЖНАЯ ИНФОРМАЦИЯ

Для данного модуля можно получить Set<Requires>, вызвав module.getDescriptor().requires(). Экземпляр Requires содержит несколько фрагментов информации, в частности имя требуемого модуля (метод name()) и необработанные и проанализированные версии, которые были скомпилированы (методы rawCompiledVersion() и compiledVersion() соответственно). В листинге 13.2 показан код, который дескриптор модуля получает, а затем передает поток по записанным директивам requires.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Листинг 13.2. Вывод информации о версии зависимости

```
module
    .getDescriptor()
    .requires().stream()
    .map(requires -> String.format("\t-> %s @ %s",
            requires.name(),
            requires.rawCompiledVersion().orElse("unknown")))
    .forEach(System.out::println);
```

Этот код производит вывод, подобный следующему:

```
> monitor @ 1.0
>     -> monitor.persistence @ 1.0
>     -> monitor.statistics @ 1.0
>     -> java.base @ 9.0.4
# остальные зависимости опущены
```

И вот они — версии зависимостей, относительно которых был скомпилирован *monitor*. Прекрасно.

Довольно просто написать класс, использующий эту информацию для сравнения версии, с которой был скомпилирован модуль, с фактической версией зависимости во время выполнения. Он может предупреждать о потенциальных проблемах, например, если фактическая версия ниже, или регистрировать всю эту информацию для последующего анализа в случае возникновения проблем.

13.3. Запуск нескольких версий модуля в отдельных слоях

В подразделе 13.1.1 говорится, что система модулей не имеет встроенной поддержки для запуска нескольких версий одного и того же модуля. Но, как я уже намекнул, это не значит, что такое невозможно. Ниже описаны способы, с помощью которых разработчики делали это до появления JPMS.

- Инструменты сборки могут *затенять* зависимости в JAR, то есть все файлы классов из зависимости копируются в целевой JAR, но под новым именем пакета. Ссылки на эти классы также обновляются, чтобы использовать новые имена классов. Таким образом, автономный JAR-файл Guava с пакетом `com.google.collect` больше не нужен, поскольку его код был перемещен в `org.library.com.google.collection`. Если каждый проект делает это, то разные версии Guava никогда не смогут конфликтовать.
- Некоторые проекты используют OSGi или другую систему модулей, которая поддерживает несколько версий «из коробки».
- Другие проекты создают собственную иерархию загрузчика классов, чтобы предотвратить конфликт различных экземпляров. (Это также то, что делает OSGi.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

У каждого из этих подходов есть недостатки, которые я не буду здесь рассматривать. Если вам абсолютно *необходимо* запустить несколько версий одного и того же JAR-файла, то потребуется найти решение, которое оправдало бы усилия, затраченные для вашего проекта.

ВАЖНАЯ ИНФОРМАЦИЯ

Тем не менее система модулей переупаковывает существующее решение, и на этом я остановлюсь в данном разделе. Можно запускать несколько версий бок о бок, однако вы увидите, что это довольно сложно и может не стоить затраченных усилий. Это не рецепт, а скорее демонстрационный пример.

13.3.1. Зачем нужен стартер для раскрутки дополнительных слоев

Как обсуждалось в разделе 12.4, система модулей представляет концепцию *слоев*, которые, по существу, связывают модульный граф с загрузчиками классов. Всегда существует хотя бы один уровень: загрузочный, который система модулей создает во время запуска на основе содержимого пути модуля.

Кроме того, слои могут быть созданы во время выполнения и нуждаются в наборе модулей в качестве входных данных: например, из каталога в файловой системе, который они затем оценивают в соответствии с правилами читабельности, чтобы гарантировать надежную конфигурацию. Поскольку слой не может быть создан, если содержит несколько версий одного и того же модуля, единственный способ сделать это — расположить их в разных слоях.

ВАЖНАЯ ИНФОРМАЦИЯ

Это значит, что вместо запуска приложения вам нужно запустить стартер, который ожидает следующий ввод:

- пути ко всем модулям приложения;
- отношения модуля, которые должны учитывать их разные версии.

Затем необходимо создать график будущих слоев, которые расположены так, что каждый слой содержит каждый модуль лишь единожды, хотя разные слои могут содержать один и тот же модуль в нескольких версиях. Последний шаг — заполнить фактические слои и затем вызвать метод main.

Разработка такого стартера в качестве общего решения является серьезной инженерной задачей и фактически означает реализацию существующих модульных систем сторонних производителей. Однако создать стартер, который решит только вашу конкретную проблему, проще, поэтому мы сосредоточимся на данном аспекте. К концу раздела вы узнаете, как создать простую структуру слоев, позволяющую запускать две версии одного и того же модуля.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

13.3.2. Раскручивание слоев для вашего приложения, Apache Twill и Cassandra Java Driver

Скажем, вы зависите от двух проектов: Apache Twill и Cassandra Java Driver. Они выдвигают противоречивые требования к версии для Guava: Apache Twill ломается при любой версии после 13, а Cassandra Java Driver — при любой версии до 16. Вы перепробовали все возможное, чтобы обойти проблему, но ничего не помогло, и теперь осталось решить проблему с помощью слоев.

Это значит, базовый слой содержит только стартер приложения. Стартер должен создать один слой с Guava 13, а другой — с Guava 16 — они должны ссылаться на базовый слой для получения доступа к платформенным модулям. Затем идет четвертый слой с остальными приложениями и зависимостями — он ссылается на другие слои, которые создает стартер, чтобы он мог искать в них зависимости.

Это не будет работать в точности с приведенным описанием. Как только зависимости Apache Twill будут разрешены, система модулей увидит Guava дважды: в каждом из слоев находятся ссылки на верхний слой. Но модулю не разрешено читать другой модуль больше одного раза, поскольку неясно, классы каких версий следует загружать.

Таким образом, вы переносите эти два модуля и все их зависимости в соответствующий слой Guava, и все готово. Почти. Оба модуля демонстрируют свою зависимость от Guava, вследствие чего ваш код должен также видеть Guava; и если он находится на верхнем уровне, то вы попадаете в ту же ситуацию, что и раньше, когда система модулей жалуется на то, что код видит две версии Guava.

Если вы перетянете свой специфичный для Twill и Cassandra код в соответствующие слои, то получите граф слоев, показанный на рис. 13.3. Теперь создадим данные слои. Для этого предположим, что вы организовали модули приложения в три каталога:

- `mods/twill` содержит Apache Twill со всеми его зависимостями и вашими модулями, которые напрямую с ним взаимодействуют (в данном примере *app.twill*);
- `mods/cassandra` содержит Cassandra Java Driver со всеми его зависимостями и вашими модулями, которые напрямую с ним взаимодействуют (в данном примере *app.cassandra*);
- `mods/app` содержит остальную часть вашего приложения и его зависимостей (в этом примере основной модуль — *app*).

Ваш стартер может затем действовать так, как показано в листинге 13.3.

1. Создайте слой с модулями в `mods/cassandra`. Будьте осторожны при выборе правильного модуля как корневого для процесса разрешения. Выберите загрузочный слой в качестве родительского.
2. Сделайте то же самое для модулей в `mods/twill`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

3. Создайте слой с модулями в `mods/app` и выберите основной модуль в качестве корневого. Используйте два других слоя как родительские; таким образом, зависимость вашего приложения от модулей в `mods/cassandra` и `mods/twill` может быть разрешена.
4. Когда все это закончится, получите загрузчик классов для основного модуля верхнего уровня и вызовите его метод `main`.

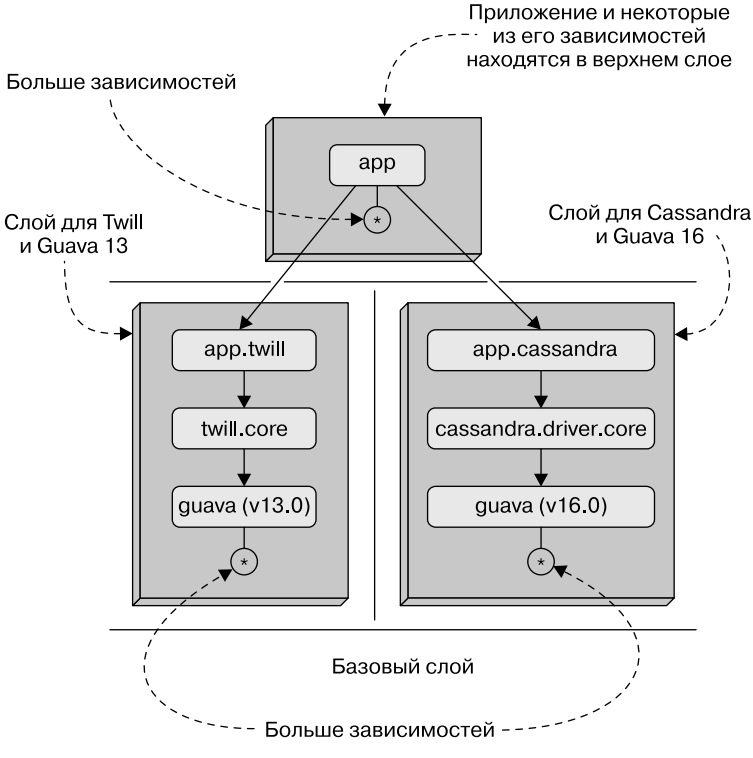


Рис. 13.3. Apache Twill и Cassandra Java Driver имеют конфликтующие зависимости от Guava. Чтобы запустить приложение с использованием обеих библиотек, каждая библиотека, включая соответствующие зависимости, должна идти в собственном слое. Над ними находится слой, содержащий остальную часть приложения, а ниже — базовый

Листинг 13.3. Стартер, создающий слои для Cassandra, Apache Twill и приложения

```
public static void main(String[] args)
        throws ReflectiveOperationException {
    createApplicationLayers()
        .findLoader("app")
        .loadClass("app.Main")
        .getMethod("main", String[].class)
        .invoke(null, (Object) new String[0]);
}
```

После того как слои приложения созданы, загружает основной класс приложения и вызывает метод main

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
private static ModuleLayer createApplicationLayers() {
    Path mods = Paths.get("mods");

    ModuleLayer cassandra = createLayer(
        List.of(ModuleLayer.boot()),
        mods.resolve("cassandra"),
        "app.cassandra");
    ModuleLayer twill = createLayer(
        List.of(ModuleLayer.boot()),
        mods.resolve("twill"),
        "app.twill");

    return createLayer(
        List.of(cassandra, twill),
        mods.resolve("app"),
        "app");
}

private static ModuleLayer createLayer(
        List<ModuleLayer> parentLayers,
        Path modulePath,
        String rootModule) {
    Configuration configuration = createConfiguration(
        parentLayers,
        modulePath,
        rootModule);
    return ModuleLayer
        .defineModulesWithOneLoader(
            configuration,
            parentLayers,
            ClassLoader.getSystemClassLoader())
        .layer();
}

private static Configuration createConfiguration( #D
        List<ModuleLayer> parentLayers,
        Path modulePath,
        String rootModule) {
    List<Configuration> configurations = parentLayers.stream()
        .map(ModuleLayer::configuration)
        .collect(toList());
    return Configuration.resolveAndBind(
        ModuleFinder.of(),
        configurations,
        ModuleFinder.of(modulePath),
        List.of(rootModule)
    );
}
```

Создает один слой для Twill и другой — для Cassandra, каждый из которых содержит весь проект плюс ваши модули, взаимодействующие с ним

Основной слой приложения начинает разрешаться в вашем главном модуле и имеет родительские слои twill и cassandra

Методы createLayer и createConfiguration аналогичны методам, приведенным в подразделе 12.4.3. Основное различие заключается в том, что они определяют корневые модули для разрешения (раньше в этом не было необходимости, поскольку мы полагались на привязку сервиса, а сейчас нет)

Вот и все! Я признаю, что описанное занимает некоторое время, и вам, вероятно, придется немного повозиться, чтобы заставить это работать (мне пришлось), но если это единственное оставшееся решение, то стоит попробовать.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Резюме

- Команды javac и `jar` позволяют записывать версии модуля с параметром `--module-version ${`*версия*`}`. Он встраивает данную версию в декларацию модуля, где ее можно прочитать с помощью командоподобных инструментов (например, `jar --describe-module`) и API рефлексии (`ModuleDescriptor::rawVersion`). Трассировка стека также показывает версии модуля.
- Если модуль знает собственную версию и другой модуль скомпилирован вместе с ним, то компилятор запишет версию в дескриптор второго модуля. Эта информация доступна только для экземпляров `Requires`, возвращаемых `ModuleDescriptor::require`.
- Система модулей никак не влияет на информацию о версии. Вместо того чтобы пытаться выбрать конкретную версию для модуля, если его путь содержит несколько, она завершает работу, выдавая сообщение об ошибке. Это исключает дорогой алгоритм выбора версии из JVM и стандарта Java.
- Система модулей не имеет встроенной поддержки для запуска нескольких версий одного и того же модуля. Основная причина — механизм загрузки классов, который предполагает, что каждый загрузчик знает не более одного класса для любого заданного имени. Если вам нужно запустить несколько версий, то требуется более одного загрузчика классов.
- OSGi делает именно это, создавая один загрузчик классов для каждого JAR-файла. Создание аналогичного общего решения — сложная задача, но возможен более простой вариант, адаптированный к вашей конкретной проблеме. Для запуска нескольких версий одного и того же модуля создайте слои и соответствующие загрузчики классов, чтобы конфликтующие модули были разделены.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

14 Настройка образа среды выполнения с помощью jlink

В этой главе

- Создание образов с выбранным контентом.
- Генерация встроенных программ запуска приложений.
- Оценка безопасности, производительности и стабильности образов.
- Генерация и оптимизация образов.

Одним из ключевых мотивов для обсуждения модульности в Java всегда было то, что сейчас называется Интернетом вещей (IoT). Это справедливо для OSGi, наиболее широко используемой сторонней системы модулей Java, которая была создана в 1999 году для улучшения разработки встроенных Java-приложений. Точно так же это справедливо и для Project Jigsaw, который спроектировал JPMS и стремился сделать платформу более масштабируемой, позволяя создание очень маленьких сред выполнения только с кодом, необходимым (встроенному) приложению.

Вот здесь и появляется `jlink`. Это инструмент командной строки Java (в папке `bin` JDK), который можно использовать для выбора ряда платформенных модулей и связывания их в образ выполнения. Такой образ действует идентично JRE, но содержит только выбранные вами модули и зависимости, которые им необходимы для работы (как указано в директивах `requires`). На данном этапе связывания `jlink` можно использовать для дальнейшей оптимизации размера образа и повышения производительности виртуальной машины, особенно времени запуска.

За годы, прошедшие с момента появления Jigsaw, многое изменилось. С одной стороны, дисковое пространство во встроенных устройствах больше не имеет такой

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ценности. В то же время мы наблюдаем рост виртуализации, особенно в Docker, где размер контейнера снова является проблемой (хотя и не основной). Рост контейнеризации также привел к необходимости облегчить и автоматизировать развертывание, что сегодня делается на несколько порядков чаще.

И здесь помогает `jlink`. Он не останавливается на связывании платформенных модулей — может создавать и образы приложений, включающие в себя код приложения, а также модули библиотек и фреймворков. Это позволяет процессу сборки создать полностью автономный модуль развертывания, состоящий из всего приложения и содержащий именно те платформенные модули, которые ему необходимы, оптимизированы для размера и производительности образа так, как вы считаете нужным, и запускаются простым вызовом собственного сценария.

Если вы в большей степени являетесь разработчиком приложений для ПК и при упоминании мною IoT и Docker ваши глаза остекленели, то этот последний момент должен был заставить вас заинтересоваться. С помощью `jlink` чрезвычайно легко распространить один Zip-файл, который пользователи могут запускать без дальнейшей настройки. И если вы использовали `javapackager`, то будете рады услышать, что теперь он вызывает `jlink` для внутренних целей, предоставляя вам доступ ко всем его функциям (хотя я не буду вдаваться в интеграцию — в документации по `javapackager` есть нужная информация).

Итак, начнем! Мы перейдем к созданию образов среды выполнения из платформенных модулей (см. раздел 14.1) и воспользуемся этой возможностью, чтобы более подробно изучить процесс компоновки, посмотреть на сгенерированный образ и обсудить, как выбрать правильные модули. Далее мы включим модули приложений и создание пользовательских модулей запуска (см. раздел 14.2), прежде чем обсудим создание образов для операционных систем (см. раздел 14.3). В завершение рассмотрим оптимизацию размера и производительности (см. раздел 14.4).

Чтобы продолжить, взгляните на ветку `feature-jlink` в репозитории *ServiceMonitor*. К концу главы вы узнаете, как создавать оптимизированные образы среды выполнения, возможно включая все приложение, для различных ОС. Это позволяет создать единый модуль развертывания, который работает «из коробки» на ваших серверах или компьютерах ваших клиентов.

14.1. Создание пользовательских образов среды выполнения

Один большой пример использования `jlink` — создание образов среды выполнения Java, которые содержат только модули, необходимые для вашего приложения. Результатом является специализированная JRE, содержащая нужные коду модули и ничего кроме них. Затем вы сможете задействовать двоичный файл Java в этом образе для запуска приложения, как и любой другой JRE.

Настройка среды выполнения имеет несколько преимуществ: можно сэкономить некое дисковое пространство (меньший образ) и, возможно, пропускную способность сети (при удаленном внедрении), приложение становится более защищенным (меньшее число классов означает меньшую поверхность атаки), и вы даже получите JVM, которая запускается немного быстрее (подробнее об этом — в подразделе 14.4.3).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ПРИМЕЧАНИЕ

С учетом сказанного jlink «просто» связывает байт-код — он не компилирует его в машинный код. Возможно, вы слышали, что, начиная с версии 9, Java экспериментирует с опережающей компиляцией (AOT), но jlink не имеет к этому никакого отношения. Чтобы узнать об AOT в Java, ознакомьтесь с Предложением по улучшению Java 295 (openjdk.java.net/jeps/295).

ВАЖНАЯ ИНФОРМАЦИЯ

Можно создавать образы среды выполнения, адаптированные к приложению, как только оно будет работать на Java 9+. Нет необходимости сначала модуляризовывать его.

Чтобы понять, как создавать образы среды выполнения с помощью `jlink`, начнем с самого простого образа (см. подраздел 14.1.1), а затем рассмотрим результат (см. подраздел 14.1.2). Далее обсудим особый подход к сервисам (см. подраздел 14.1.3), прежде чем завершить раздел реальным примером использования: как создать образ, предназначенный для запуска данного приложения (см. подраздел 14.1.4).

14.1.1. Начало работы с jlink

Определение: необходимая информация для jlink

Для создания образа `jlink` требуется *три компонента*, каждый из которых указывается с помощью параметра командной строки:

- где найти доступные модули (указывается с `--module-path`);
- какие модули использовать (указывается с `--add-modules`);
- папка, в которой создается образ (указывается с помощью `--output`).

Простейший возможный образ среды выполнения содержит только базовый модуль. Листинг 14.1 показывает, как его создать с помощью `jlink`.

Листинг 14.1. Создание образа среды выполнения, содержащего только базовый модуль

```
$ jlink
    --module-path ${jdk-9}/jmods
    --add-modules java.base
    --output jdk-base
$ jdk-base/bin/java --list-modules

> java.base
```

Расположение модулей, в этом случае платформенных из локальной установки JDK

Модули для добавления к образу, в данном случае только java.base

Выходной каталог для образа

Выполняет java --list-modules из нового образа с целью убедиться в том, что он содержит только базовый модуль

Может показаться немного странным, что нужно указывать `jlink`, где искать платформенные модули. Это не требуется для `javac` и `java`, так почему `jlink` не зна-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ет, где их найти? Ответ — кросс-платформенное соединение, которое обсуждается в разделе 14.3.

ПРИМЕЧАНИЕ

Начиная с Java 10, больше нет необходимости размещать модули платформы в пути модулей. Если он не содержит ничего, то jlink неявно загружает их из каталога $JAVA_HOME/jmods.

ВАЖНАЯ ИНФОРМАЦИЯ

Независимо от того, ссылаются ли платформенные модули явным или неявным образом, рекомендуется загружать их только из той же версии JVM, что и двоичный файл jlink. Например, если jlink имеет версию 9.0.4, то убедитесь, что он загружает модули платформы из JDK 9.0.4.

Учитывая три полученных параметра командной строки, `jlink` разрешает модули, как описано в подразделе 3.4.1: содержимое пути модуля становится совокупностью обозреваемых модулей, а модули, передаваемые `--add-modules`, — корневыми для процесса разрешения. Но у `jlink` есть несколько особенностей.

ВАЖНАЯ ИНФОРМАЦИЯ

По умолчанию сервисы (см. главу 10) не связаны. Подраздел 14.1.3 объясняет почему и исследует, что с этим делать.

- Необязательные зависимости с requires static (см. раздел 11.2) не разрешаются. Их нужно добавлять вручную.
- Автоматические модули не допускаются. Этот аспект становится важным в разделе 14.2 и там же объясняется более подробно.

Если не возникнут какие-либо проблемы, такие как отсутствующие или дублирующие модули, то разрешенные модули (корневые модули плюс транзитивные зависимости) окажутся в новом образе среды выполнения. Посмотрим на это.

14.1.2. Содержание и структура образа

Прежде всего, данный образ занимает всего около 45 Мбайт (в Linux; я слышал, что в Windows он еще меньше) по сравнению с 263 Мбайт полной JRE — и это без оптимизации пространства, описанной в подразделе 14.4.2. Так как же выглядит образ? В разделе 6.3 представлена новая структура каталогов JDK/JRE; и как показано на рис. 14.1, образы среды выполнения, созданные с помощью `jlink`, похожи. Это не совпадение: JDK и JRE, которые можно загрузить, состоят из `jlink`.

Обратите внимание: `jlink` объединяет включенные модули в `lib/modules`, а затем убирает папку `jmods` из окончательного образа. Это соответствует тому, как был сгенерирована JRE, которая тоже не содержит `jmods`. Необработанные файлы JMOD включены только в JDK, так что `jlink` может их обрабатывать: оптимизация модулей в `lib/modules` является односторонней операцией, а `jlink` не может генерировать дополнительные образы из оптимизированного образа.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

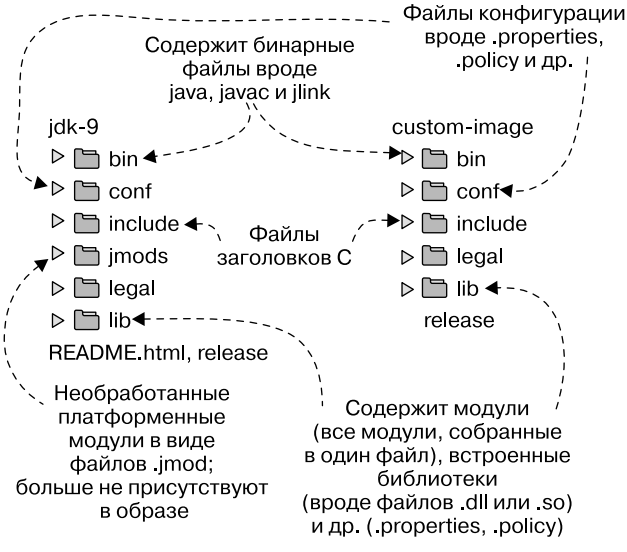


Рис. 14.1. Сравнение структуры каталогов JDK *(слева)* и пользовательского образа среды выполнения, созданного с помощью jlink *(справа)*. Сходство неслучайно — JDK создается с помощью jlink

Заглянув в `bin`, вы можете спросить, какие исполняемые файлы ожидаете там найти. Оказывается, `jlink` умный и станет генерировать только исполняемые файлы, для которых необходимые модули были включены в образ. Например, исполняемый файл компилятора `javac` поставляется с модулем *jdk.compiler* и не будет доступен, если этот модуль не включен.

14.1.3. Включение сервисов в образы среды выполнения

Если внимательно посмотреть на листинг 14.1, то может показаться странным, что изображение содержит только *java.base*. В подразделе 10.2.1 вы узнали, что базовый модуль использует множество сервисов, предоставляемых другими платформенными модулями, и когда сервисы связаны во время разрешения модуля, все эти поставщики включаются в модульный граф. Так почему они также не попадают в образ?

Определение: --bind-services

Чтобы разрешить создание небольших, преднамеренно собранных образов среды выполнения, `jlink` по умолчанию не производит привязку сервисов при создании образа. Вместо этого модули поставщиков сервисов, перечисленные в `--add-modules`, должны быть добавлены вручную. В качестве альтернативы *параметр* `--bind-services` *может применяться для включения всех модулей, которые предоставляют сервис*, используемый другим разрешенным модулем.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В качестве примера выберем кодировки, такие как ISO-8859-1, UTF-8 или UTF-16. Базовый модуль знает, что вам нужно ежедневно, но есть специальный платформенный модуль, который содержит несколько других: *jdk.charsets*. Базовый модуль и *jdk.charsets* отделены через сервисы. Вот соответствующие части их модульных деклараций:

```
module java.base {
    uses java.nio.charset.spi.CharsetProvider;
}

module jdk.charsets {
    provides java.nio.charset.spi.CharsetProvider
        with sun.nio.cs.ext.ExtendedCharsets
}
```

Когда JPMS разрешает модули во время обычного запуска, привязка сервисов будет тянуться в *jdk.charsets*, так что *charsets* не всегда были доступны в стандартной JRE. Но при создании образа среды выполнения с помощью `jlink` такого не случается, вследствие чего по умолчанию образы не содержат модуль *charsets*. Если ваш проект зависит от этого, то вы можете столкнуться с трудностями.

Определив свою зависимость от модуля, который отделен от остальных с помощью сервисов, вы можете добавить его в образ с помощью `--add-modules`:

```
$ jlink
    --module-path ${jdk-9}/jmods
    --add-modules java.base,jdk.charsets
    --output jdk-charsets
$ jdk-charsets/bin/java --list-modules

> java.base
> jdk.charsets
```

Определение: --suggest-providers

Определять модули поставщиков сервисов вручную может быть обременительно. К счастью, на помощь приходит `jlink`. Параметр `--suggest-provider ${`*`сервис`*`}` *перечисляет все обозреваемые модули, обеспечивающие реализацию* `${`*`сервис`*`}`, которая должна быть указана с полным именем.

Допустим, вы создали минимальный образ среды выполнения, содержащий только *java.base*, и при запуске приложения возникают проблемы из-за отсутствия кодировок. Вы отслеживаете проблему в *java.base*, используя `java.nio.charset.spi.CharsetProvider`, и теперь задаетесь вопросом, какие модули предоставляют этот сервис. Вот здесь и пригодится `--suggest-providers`:

```
$ jlink
    --module-path ${jdk-9}/jmods
    --suggest-providers java.nio.charset.spi.CharsetProvider
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
> Suggested providers:
> jdk.charsets
>     provides java.nio.charset.spi.CharsetProvider
>     used by java.base
```

Еще один хороший пример для скрытых модулей — языковые стандарты. Все стандарты, кроме английских, содержатся в *jdk.localedata*, что делает их доступными для базового модуля через сервис. Рассмотрим следующий код:

```
String half = NumberFormat
    .getInstance(new Locale("fi", "FI"))
    .format(0.5);
System.out.println(half);
```

Что он выведет? `Locale("fi", "FI")` создает языковой стандарт для Финляндии, а в финском формате используются числа с плавающей точкой, но с запятой, поэтому результат будет равен `0,5` — по крайней мере если доступен финский языковой стандарт. Запустив этот код в образе выполнения, который не содержит *jdk.localedata*, например, в созданном ранее, вы получите `0.5`, поскольку Java молча возвращается к языку по умолчанию. Да, это не ошибка, а бесшумное поведение.

Как и раньше, решение состоит в том, чтобы явно включить разъединенный модуль, в данном случае *jdk.localedata*. Однако он добавляет колоссальные 16 Мбайт к размеру образа, поскольку содержит много региональных данных. К счастью, как вы увидите в подразделе 14.4.2, `jlink` может помочь уменьшить эту дополнительную нагрузку.

ПРИМЕЧАНИЕ

> Если поведение приложения в загруженном общедоступном Java отличается от запуска в настраиваемом образе среды выполнения, то следует подумать о сервисах. Может ли неправильное поведение быть вызвано недоступностью какой-либо функции JVM? Возможно, нужный модуль был отделен с помощью сервисов и теперь отсутствует в образе выполнения.

Ниже представлены отдельные сервисы, используемые базовым модулем и другими платформенными модулями, от которых вы можете неявно зависеть:

- кодировки из `jdk.charsets`;
- языковые стандарты из `jdk.localedata`;
- файловая система Zip из `jdk.zipfs`;
- поставщики сервисов безопасности из *java.naming*, *java.security.jgss*, *java.security.sasl*, *java.smartcardio*, *java.xml.crypto*, *jdk.crypto.cryptoki*, *jdk.crypto.ec*, *jdk.deploy* и *jdk.security.jgss*.

В качестве альтернативы идентификации и добавлению отдельных модулей вручную можно использовать параметр `--bind-services`:

```
$ jlink
    --module-path ${jdk-9}/jmods
    --add-modules java.base
    --bind-services
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
    --output jdk-base-services
$ jdk-base-services/bin/java --list-modules

> java.base
> java.compiler
> java.datatransfer
> java.desktop
# опущено еще около тридцати модулей
```

Тем не менее это связывает все модули, которые предоставляют сервис с базовым модулем, и, таким образом, создает довольно большой образ — его размер составляет около 150 Мбайт без оптимизации. Стоит хорошенько взвесить все за и против, прежде чем выбирать такой путь.

14.1.4. Правильно подобранные образы с помощью jlink и jdeps

До сих пор вы создавали только небольшие образы, состоящие из *java.base* и нескольких других модулей. Но как насчет реального варианта использования; как бы вы определили, какие платформенные модули нужны для поддержки большого приложения? Здесь нельзя использовать метод проб и ошибок, верно?

Ответ — JDeps. Подробное введение см. в приложении Г. Сейчас же достаточно знать, что в следующем заклинании перечислены все платформенные модули, от которых зависит приложение:

```
jdeps -summary -recursive --class-path 'jars/*' jars/app.jar
```

Чтобы это работало, папка `jars` должна включать все JAR-файлы, необходимые для запуска приложения (код, а также зависимости; для этого пригодится инструмент сборки), а `jars/app.jar` должен содержать основной метод, который вы используете при запуске. Результат покажет множество зависимостей между артефактами, но в том числе и строки, показывающие зависимости от платформенных модулей. В следующем примере перечислены платформенные модули, применяемые Hibernate Core 5.2.12, и его зависимости:

```
antlr-2.7.7.jar -> java.base
classmate-1.3.0.jar -> java.base
dom4j-1.6.1.jar -> java.base
dom4j-1.6.1.jar -> java.xml
hibernate-commons-annotations-5.0.1.Final.jar -> java.base
hibernate-commons-annotations-5.0.1.Final.jar -> java.desktop
hibernate-core-5.2.12.Final.jar -> java.base
hibernate-core-5.2.12.Final.jar -> java.desktop
hibernate-core-5.2.12.Final.jar -> java.instrument
hibernate-core-5.2.12.Final.jar -> java.management
hibernate-core-5.2.12.Final.jar -> java.naming
hibernate-core-5.2.12.Final.jar -> java.sql
hibernate-core-5.2.12.Final.jar -> java.xml
hibernate-core-5.2.12.Final.jar -> java.xml.bind
hibernate-jpa-2.1-api-1.0.0.Final.jar -> java.base
hibernate-jpa-2.1-api-1.0.0.Final.jar -> java.instrument
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
hibernate-jpa-2.1-api-1.0.0.Final.jar -> java.sql
jandex-2.0.3.Final.jar -> java.base
javassist-3.22.0-GA.jar -> java.base
javassist-3.22.0-GA.jar -> jdk.unsupported
jboss-logging-3.3.0.Final.jar -> java.base
jboss-logging-3.3.0.Final.jar -> java.logging
slf4j-api-1.7.13.jar -> java.base
```

Все, что нужно сделать сейчас, — извлечь эти строки, удалить часть … -> и выбросить дубликаты. Для пользователей Linux:

```
jdeps -summary -recursive --class-path 'jars/*' jars/app.jar
    | grep '\-> java.\|\-> jdk.'
    | sed 's/^.*-> //'
    | sort -u
```

В итоге вы получите точный список платформенных модулей, от которых зависит приложение. Добавьте их в `jlink --add-modules` и получите наименьший возможный образ среды выполнения, который поддерживает ваше приложение (рис. 14.2).

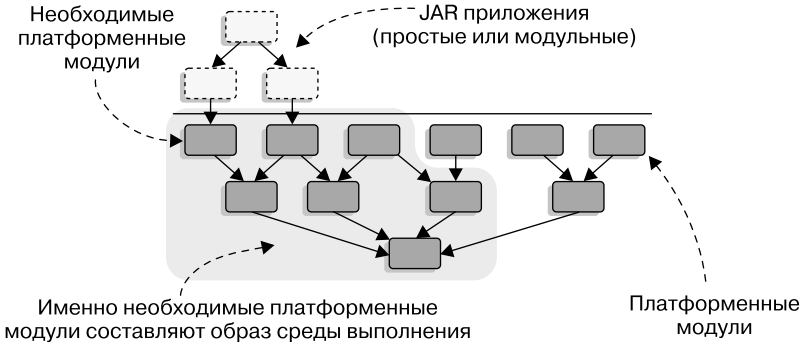


Рис. 14.2. С учетом JAR приложения *(сверху)* и их зависимостей от платформенных модулей *(снизу)* jlink может создать образ среды выполнения только при наличии нужных платформенных модулей

ВАЖНАЯ ИНФОРМАЦИЯ

Невзирая на все это, есть несколько предостережений.

- JDeps иногда сообщает, что … -> not found, то есть какая-то переходная зависимость не была найдена в пути к классам. Убедитесь, что путь к классам для JDeps содержит точные артефакты, которые используются при запуске приложения.
- JDeps не может анализировать рефлексию, поэтому в случае взаимодействия вашего кода или кода ваших зависимостей с классами JDK только с помощью рефлексии JDeps не распознает данное обстоятельство. Это может привести к тому, что требуемый модуль не попадет в образ.
- Как обсуждалось в подразделе 14.1.3, jlink не связывает сервисы по умолчанию, но ваше приложение может неявно полагаться на присутствие некоторых внутренних поставщиков JDK.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- Рассмотрите возможность добавления модуля java.instrument, который требуется для поддержки агентов Java. Это необходимо, если ваша производственная среда использует агенты для наблюдения за работающими приложениями. Но даже если это не так, вы можете оказаться в затруднительном положении, когда применение Java-агента — лучший способ проанализировать проблему. Кроме того, это всего лишь около 150 дополнительных килобайт, которые вряд ли имеют большое значение.

ПРИМЕЧАНИЕ

После создания образа среды выполнения для приложения я рекомендую запустить модульные и интеграционные тесты. Это позволит вам быть уверенными в том, что вы действительно добавили все необходимые модули.

Следующий шаг — включение модулей приложений в образ, но для этого приложение и его зависимости должны быть полностью модульными. Если это не так и вы ищете более оперативно применимые знания, то перейдите к разделу 14.3 для генерации образов среды выполнения для разных ОС или к разделу 14.4 для оптимизации образа.

14.2. Создание автономных образов приложений

До сих пор вы создавали образы среды выполнения, *поддерживающие* приложение, однако нет причин останавливаться на этом. `jlink` позволяет легко создавать образы, *содержащие* все приложение. Это значит, что вы получите образ, содержащий модули приложения (само приложение и его зависимости) и платформенные модули, необходимые для его поддержки. Вы даже можете создать красивое средство запуска, чтобы запустить приложение с помощью `bin/my-app`! Распространять приложение теперь станет намного проще.

Определение: образы приложений

Пусть это и не официальный термин, я называю образы, включающие модули приложений, образами приложений (в отличие от образов среды выполнения), чтобы четко обозначить то, о чем говорю. В конце концов, результат больше похож на приложение, чем на общее выполнение.

ВАЖНАЯ ИНФОРМАЦИЯ

Обратите внимание, что jlink работает только с явными модулями, поэтому приложение, зависящее от автоматических модулей (см. раздел 8.3), не может быть связано с образом. Если же абсолютно необходимо создать образ для такого приложения, то ознакомьтесь с подразделом 9.3.3, в котором сказано, как сделать сторонние JAR-файлы модульными, или используйте инструмент, например ModiTect (github.com/moditect/moditect), который сделает это за вас.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Данное ограничение явных модулей не имеет технических оснований — это было дизайнерское решение. Предполагается, что образ приложения является автономным, но если он зависит от автоматических модулей, которые не выражают зависимости, JPMS не может это проверить, и потому может возникнуть ошибка `NoClassDefFoundError`. Не совсем та надежность, к которой стремится система модулей.

Начнем, учитывая эти предварительные условия. Сначала вы создадите образ, который включает в себя модули приложения (см. подраздел 14.2.1), а затем упростите свою жизнь, создав средства запуска (см. подраздел 14.2.2). Наконец, мы рассмотрим безопасность, производительность и стабильность образов приложений (см. подраздел 14.2.3).

14.2.1. Включение модулей приложений в образы

Все, что нужно сделать для создания образа приложения, — это добавить модули приложения в путь модуля `jlink` и выбрать один или несколько из них в качестве корневых. Полученный образ будет содержать все необходимые модули (но никаких других, кроме этих; рис. 14.3) и может быть запущено с помощью `bin/java --module ${`*`начальный_модуль`*`}`.

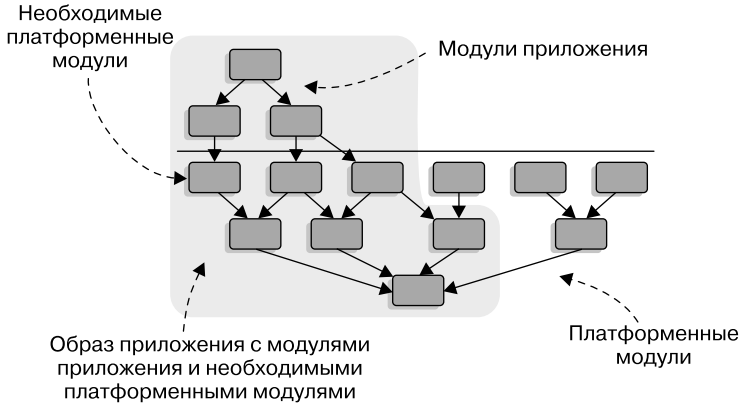


Рис. 14.3. С учетом модулей приложения *(сверху)* и их зависимостей от платформенных модулей *(снизу)* jlink может создать образ среды выполнения только с необходимыми модулями, включая код приложения и платформы

В качестве примера еще раз обратимся к приложению *ServiceMonitor*. Поскольку оно зависит от автоматических модулей *spark.core* и *hibernate.jpa*, а `jlink` их не поддерживает, мне пришлось отключить эту функцию. Таким образом, остается семь модулей, каждый из которых зависит только от *java.base*:

- *monitor*;
- *monitor.observer*;
- *monitor.observer.alpha*;
- *monitor.observer.beta*;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- *monitor.persistence*;
- *monitor.rest*;
- *monitor.statistics*.

Я поместил их в папку с именем `mods` и создал образ, как показано в листинге 14.2. К сожалению, я забыл, что реализации обозревателя *monitor.observer.alpha* и *monitor.observer.beta* отделены от остальной части приложения с помощью сервисов и что они не связаны по умолчанию (см. главу 10, посвященную сервисам, и подраздел 14.1.3, описывающий, как `jlink` их обрабатывает). Поэтому мне пришлось повторить попытку в листинге 14.3, добавив их явно. В качестве альтернативы я мог бы использовать `--bind-services`, но мне не понравилось то, насколько большим стал образ, когда в него были добавлены все внутренние поставщики сервисов JDK.

Листинг 14.2. Создание образа приложения, содержащего ServiceMonitor

```
$ jlink
    --module-path ${jdk-9}/jmods:mods
    --add-modules monitor
    --output jdk-monitor
$ jdk-monitor/bin/java --list-modules

> java.base
> monitor
> monitor.observer
> monitor.persistence
> monitor.rest
> monitor.statistics
```

Помимо платформенных модулей, я указал модули приложений в mods. Для Windows используйте ; вместо :

Начинает разрешение модулей с monitor

Реализации сервисов monitor.observer.alpha и monitor.observer.beta отсутствуют

Листинг 14.3. Создание образа приложения, на этот раз с сервисами

```
$ jlink
    --module-path ${jdk-9}/jmods:mods
    --add-modules monitor,
        monitor.observer.alpha,monitor.observer.beta
    --output jdk-monitor
$ jdk-monitor/bin/java --list-modules

> java.base
> monitor
> monitor.observer
> monitor.observer.alpha
> monitor.observer.beta
> monitor.persistence
> monitor.rest
> monitor.statistics
```

Запускает разрешение модулей с начальным модулем (monitor) и всеми необходимыми сервисами (два других модуля)

Определение: системные модули

В совокупности платформенные модули и модули приложения, которые включены в образ, называются *системными*. Как вы скоро увидите, все еще можно добавлять другие модули при запуске приложения.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Остерегайтесь особенностей разрешения!

Напомню: в разделе 14.1 сказано, что `jlink` создает минимальный образ:

- не связывает сервисы;
- не включает необязательные зависимости.

ВАЖНАЯ ИНФОРМАЦИЯ

Хотя вы, вероятно, не забудете проверить наличие собственных сервисов, все-таки можно забыть о зависимостях (например, о реализации драйвера SQL) или о платформенных модулях (данные о языковых стандартах или необычные кодировки). То же самое касается необязательных зависимостей, которые можно добавить, но забыть, что они не разрешены только из-за их присутствия в пути модуля (см. подраздел 11.2.3). Убедитесь, что действительно добавили все необходимые модули!

Приложение *ServiceMonitor* использует финский язык для форматирования своего вывода, поэтому ему необходимо добавить *jdk.localedata* к образу (листинг 14.4). Это увеличивает размер изображения на 16 Мбайт (до 61 Мбайт), но в подразделе 14.4.2 показано, как его уменьшить.

Листинг 14.4. Создание образа приложения ServiceMonitor с данными языкового стандарта

```
$ jlink
    --module-path ${jdk-9}/jmods:mods
    --add-modules monitor,
        monitor.observer.alpha,monitor.observer.beta,
        jdk.localedata
    --output jdk-monitor
```

Платформенные модули для языковых стандартов также добавлены в образ

Использование параметров командной строки при запуске приложения

После создания образа можно запустить приложение, как обычно, с помощью `java --module ${`*`начальный_модуль`*`}`, используя исполняемый файл `java` в папке `bin` образа. Но, поскольку вы включили модули приложения в образ, вам не нужно указывать путь модуля — JPMS найдет их внутри образа.

После создания образа *ServiceMonitor* в `jdk-monitor` приложение можно запустить с помощью короткой команды:

```
$ jdk-monitor/bin/java --module monitor
```

При желании вы *можете* использовать путь модуля. В этом случае имейте в виду: системные модули (те, что в образе) всегда будут скрывать модули с одинаковыми именами в пути модуля — так, словно бы они не существовали в пути. Что можно сделать с помощью пути к модулю, так это добавить *новые* модули в приложение. Скорее всего, это будут дополнительные поставщики сервисов, которые дают воз-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

можность отправлять образ вместе с приложением, в то же время позволяя пользователям легко расширять его локально.

Допустим, *ServiceMonitor* обнаруживает новый вид микросервиса, который он должен наблюдать, и модуль *monitor.observer.zero* как раз этим и занимается. Более того, модуль реализует все нужные интерфейсы, а его дескриптор объявляет, что он предоставляет `ServiceObserver`. Затем, как показано в листинге 14.5, вы можете использовать тот же образ из предыдущего и добавить *monitor.observer.zero* через путь модуля.

Листинг 14.5. Запуск образа приложения с дополнительным поставщиком сервисов

Располагает поставщик сервиса в пути модуля

```
$ jdk-monitor/bin/java
    --module-path mods/monitor.observer.zero.jar
    --show-module-resolution
    --dry-run
    --module monitor
```

Вместо выполнения действий по действительному запуску приложения посмотрите на разрешение модуля с целью увидеть, что поставщик добавлен (и эти параметры работают как в обычной JRE)

```
> root monitor jrt:/monitor
# зависимости модуля monitor опущены
> monitor binds monitor.observer.alpha jrt:/monitor.observer.alpha
> monitor binds monitor.observer.beta jrt:/monitor.observer.beta
> monitor binds monitor.observer.zero file://...
```

Строка jrt: показывает, что эти модули загружены из образа

Дополнительный модуль загружается из пути модуля, как указано в file:

ВАЖНАЯ ИНФОРМАЦИЯ

Если вы хотите заменить системные модули, то нужно поместить их в обновленный путь модуля, как описано в подразделе 6.1.3. Вдобавок к особому случаю пути модуля все другие параметры Java, представленные в этой книге, работают точно так же в пользовательских образах приложений.

14.2.2. Создание собственного средства запуска приложения

Если создание образа, содержащего приложение и все необходимые компоненты, но не более, — торт, то добавление пользовательского средства запуска — это глазурь. Пользовательское средство запуска — исполняемый сценарий (оболочка для ОС на Unix, пакет в Windows) в папке `bin` образа, который предварительно настроен для запуска JVM с конкретным модулем и основным классом.

Как показано в листинге 14.6, с помощью `--launcher run-monitor=monitor` можно приказать `jlink` создать сценарий `run-monitor` в `bin`, который запускает приложение с эквивалентом `java --module monitor`. Поскольку *monitor* объявляет основной класс

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

(`monitor.Main`), нет необходимости указывать его с помощью `--launcher`. Захоти вы, это было бы `--launcher run-monitor=monitor/monitor.Main`.

Определение: --launcher

Чтобы создать средство запуска, используйте *параметр* `--launcher ${`*имя*`}=${`*модуль*`}/${`*основной_класс*`}`:

- `${`*имя*`}` — имя файла, выбранное для исполняемого файла;
- `${`*модуль*`}` — имя модуля для запуска;
- `${`*основной_класс*`}` — имя основного класса модуля.

Последние два — то, что вы обычно ставите после `java --module`. И как в этом случае, если модуль определяет основной класс, то можно опустить `/${`*основной_класс*`}`.

Листинг 14.6. Создание образа приложения с помощью средства запуска (и взгляд на него)

```
$ jlink
    --module-path ${jdk-9}/jmods:mods
    --add-modules monitor,
        monitor.observer.alpha,monitor.observer.beta
    --output jdk-monitor
    --launcher run-monitor=monitor
$ cat jdk-monitor/bin/run-monitor

> #!/bin/sh
> JLINK_VM_OPTIONS=
> DIR=`dirname $0`
> $DIR/java $JLINK_VM_OPTIONS -m monitor/monitor.Main $@

$ jdk-monitor/bin/run-monitor
```

Создает образ, как в листинге 14.3…

… кроме добавления модуля запуска с именем run-monitor, который запускает модуль monitor (определяющий основной класс)

Ради интереса заглядывает в сценарий (cat выводит содержимое файла)

Указывает на то, что это сценарий оболочки

Команда, которая запускается при вызове сценария

Как использовать средство запуска

ПРИМЕЧАНИЕ

Вы заметили JLINK_VM_OPTIONS в листинге 14.6? Если есть какие-либо параметры командной строки, которые вы хотели бы указать для приложения — например, для настройки сборщика мусора, — то можно указать их здесь.

Однако использование панели запуска имеет свои недостатки: все параметры, которые вы пытаетесь применить к запускаемой JVM, будут интерпретироваться так, словно вы указали их после параметра `--module`, что делает их аргументами программы. То есть при использовании средства запуска вы не можете настраивать систему модулей, например, для добавления дополнительных сервисов, как обсуждалось ранее.

Но у меня есть хорошие новости: не обязательно задействовать средство запуска, и команда `java` все еще доступна. Листинг 14.5 работает точно так же, если было создано средство запуска, — до тех пор пока вы им не пользуетесь.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

14.2.3. Безопасность, производительность и стабильность

Создание образа приложения может повысить безопасность приложения, сводя к минимуму объем кода, доступного в JVM, и, таким образом, уменьшая поверхность атаки. Как показано в подразделе 14.4.3, можно ожидать и небольших улучшений во времени запуска.

Это звучит довольно аккуратно, однако в полной мере относится только к ситуациям, когда у вас есть полный контроль над работой приложения и регулярным повторным развертыванием. Если вы отправляете образ клиентам или никак иначе не можете контролировать, когда и как часто он заменяется на новый, то таблицы переворачиваются.

ВАЖНАЯ ИНФОРМАЦИЯ

Образ, созданный с помощью jlink, не предназначен для изменений. У него нет функции автообновления, и его исправление вручную не является реалистичным сценарием. Если пользователи обновят свой системный Java, то это не повлияет на образ вашего приложения. Образ и точная версия Java, с которой были добавлены платформенные модули во время компоновки, связаны вместе навсегда.

Хорошо здесь то, что обновления Java не могут сломать ваше приложение, но гораздо более серьезным недостатком является то, что приложение не получит каких-либо исправлений безопасности или улучшений производительности, которые принесет новая версия Java. Все это будет накапливаться. Если критическая уязвимость исправлена в новой версии Java, то ваши пользователи будут по-прежнему уязвимы, пока не развернут новый образ приложения, отправленный вами.

ПРИМЕЧАНИЕ

Если вы решили распространить образы приложений, то я рекомендую сделать данный механизм дополнительным, а не единственным. Позвольте пользователям решать, хотят ли они развернуть весь образ или предпочитают запускать JAR в собственной среде выполнения, над которой они имеют полный контроль и которую могут обновлять независимо.

14.3. Генерация образов в операционных системах

Хотя байт-код, содержащийся в вашем JAR-приложении и библиотеке, не зависит от какой-либо ОС, для его выполнения требуется виртуальная машина Java (JVM), специфичная для данной ОС. Вот почему вы скачиваете JDK и среды выполнения специально для Linux, macOS или Windows (к примеру). Важно понимать, что `jlink` работает на уровне ОС! На рис. 14.4 показаны части, специфичные для ОС.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

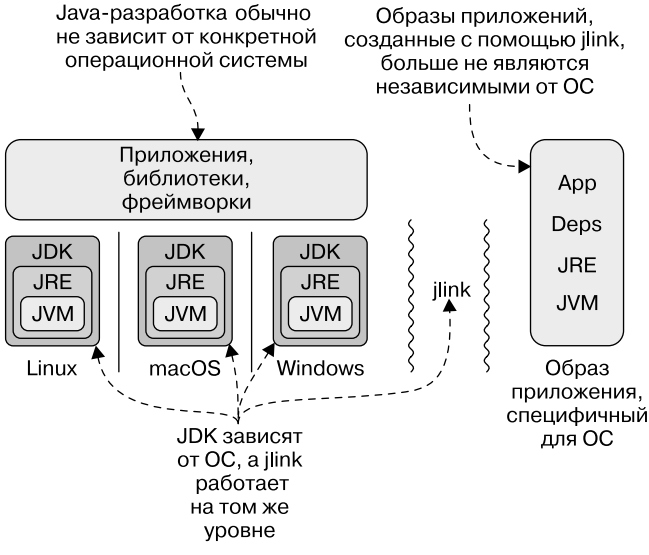


Рис. 14.4. В отличие от JAR приложений, библиотек и фреймворков *(сверху)* образы приложений *(справа)* зависят от ОС, как и JVM *(снизу)*

Это очевидно, если подумать: платформенные модули, которые `jlink` использует для создания образа, приходят из JDK/JRE для конкретной ОС, вследствие чего полученный образ тоже зависит от ОС. Иными словами, образы среды выполнения или приложения всегда привязаны к одной конкретной ОС.

Означает ли это необходимость выполнить `jlink` на нескольких разных машинах, чтобы создать все требуемые образы среды выполнения или приложения? К счастью, нет. Как вы видели в подразделе 14.1.1, при создании образа нужно указывать ссылку на платформенные модули, которые вы хотите добавить в него. Вот в чем дело: это не обязательно для ОС, в которой вы запускаете `jlink`!

ВАЖНАЯ ИНФОРМАЦИЯ

Если вы загрузите и распакуете JDK для другой ОС, то сможете поместить его папку jmods в путь модуля при запуске версии jlink из системного JDK. Затем компоновщик определит, что образ должен быть создан для этой ОС, и, следовательно, создаст тот, который работает именно в ней (но, конечно, не в какой-либо другой). Таким образом, учитывая JDK для всех ОС, поддерживаемых вашим приложением, можно генерировать образы среды выполнения или приложения для каждой из них на одном компьютере.

Я использую Linux, но, например, хочу создать образ приложения для приложения *ServiceMonitor*, работающего в macOS. Удобно, что `jlink` очень хорошо поддерживает такие сценарии, — всего-то и надо, что JDK для целевой ОС.

Оказывается, самое сложное — распаковать JDK в ОС, для которой он не был упакован. В этом случае нужно попасть в файл `*.dmg`, который Oracle распространяет для

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

macOS, — здесь я не буду вдаваться в подробности, но можно найти советы по каждой из нетривиальных комбинаций `{Linux, macOS, Windows} versus {rpm/tar.gz, dmg, exe}` в выбранной поисковой системе. В конце концов, я получил MacOS JDK в какой-то папке, который я буду представлять как `${jdk-9-mac-os}`.

Затем все, что мне нужно сделать, — то же самое, что и в подразделе 14.2.1, за исключением замены папки JDK 9 моей машины (`${jdk-9}`) на папку, содержащую JDK macOS (`${jdk-9-mac-os}`). Это значит, что я использую исполняемый файл jlink из моего Linux JDK с каталогом `jmods` в macOS JDK:

```
$ jlink
    --module-path ${jdk-9-mac-os}/jmods:mods
    --add-modules monitor,
        monitor.observer.alpha,monitor.observer.beta
    --output jdk-monitor
    --launcher run-monitor=monitor
```

Мой босс должен быть доволен. (Но если что-то пойдет не так, то я даже не смогу утверждать, будто это работает на моей машине!)

14.4. Использование плагинов jlink для оптимизации образов

«Сделайте так, чтобы это работало, сделайте это правильно, сделайте это быстро», — сказал Кент Бек, создатель экстремального программирования и автор книги *Test-Driven Development: By Example* (O'Reilly, 2000)[1]. И, рассмотрев основные аспекты создания образов среды выполнения и приложений (даже в разных ОС), мы обратимся к оптимизации. Она может значительно уменьшить размер образа и немного увеличить производительность во время выполнения, особенно во время запуска.

В `jlink` оптимизации обрабатываются плагинами. Следовательно, имеет смысл сначала поговорить о данной архитектуре плагина (см. подраздел 14.4.1), прежде чем делать образы меньше (см. подраздел 14.4.2) и быстрее (см. подраздел 14.4.3).

14.4.1. Плагины для jlink

Центральный аспект `jlink` — его модульная конструкция. Помимо необходимых шагов по определению правильных модулей и генерации образов для них, jlink оставляет дальнейшую обработку контента образа своим плагинам. Вы можете просмотреть доступные плагины с помощью `jlink --list-plugins`, см. docs.oracle.com/javase/9/tools/jlink.htm для официально поддерживаемых, или проверить табл. 14.1 на выбор (мы рассмотрим каждый из них в подразделах 14.4.2 и 14.4.3).

[1] *Бек К.* Экстремальное программирование: разработка через тестирование. Библиотека программиста. — СПб.: Питер, 2020.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Таблица 14.1. Алфавитная таблица некоторых плагинов jlink, показывающая, уменьшают ли они в основном размер образа или улучшают производительность во время выполнения

Имя	Описание	Размер	Производительность
class-for-name	Заменяет Class :: forName статическим доступом	—	✔
compress	Распределяет строковые литералы и сжимает lib/modules	✔	—
exclude-files	Исключает файлы, например встроенные бинарные файлы	✔	—
exclude-resources	Исключает ресурсы, например, из папок META-INF	✔	—
generate-jli-classe	Прегенерирует обработчики метода	—	✔
include-locales	Удаляет все, кроме указанных языковых стандартов из jdk.localedata	✔	—
order-resources	Упорядочивает ресурсы в lib/modules	—	✔
strip-debug	Удаляет символы отладки из байт-кода образа	✔	—
system-modules	Подготавливает граф системного модуля для быстрого доступа	—	✔

ПРИМЕЧАНИЕ

В документации, как и в самом jlink, также указан плагин vm, позволяющий выбрать одну из нескольких виртуальных машин HotSpot (клиентская, серверная или минимальная), которые вы хотите включить в образ. Однако это теоретическая возможность, поскольку 64-битные JDK поставляются только с виртуальной машиной сервера. Для большинства ситуаций вам придется выбрать одну из них.

Разработка плагинов для jlink

На момент печати книги доступны только поддерживаемые плагины, но это положение может измениться в дальнейшем, по добавлении дополнительных экспериментальных функций. Усилия по оптимизации изображений при их создании были приложены не так давно, и большая работа все еще продолжается. Как следствие, API плагина может измениться и не будет стандартизирован или экспортирован в Java 9+.

Это делает разработку плагинов для `jlink` довольно сложной[1] и означает, что придется подождать некоторое время, прежде чем сообщество действительно начнет добавлять плагины. Что они могли бы сделать? Прежде всего, написание плагинов `jlink` немного похоже на пишущие агенты или плагины инструментов сборки, а не на то, что делалось во время типичной разработки приложений. Это специализированная задача поддержки конкретных библиотек, фреймворков и инструментов.

[1] См. сообщение в блоге Гуннара Морлинга «Изучение API подключаемого модуля jlink в Java 9» (mng.bz/xJ6B), если вы заинтересованы в пошаговом руководстве.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Но вернемся к вопросу о том, что могут сделать предоставленные сообществом плагины. Один вариант использования исходит от профилировщиков, которые в настоящее время задействуют агенты для внедрения в работающие приложения кода, отслеживающего производительность. С помощью плагина `jlink` вы могли бы сделать это во время соединения вместо того, чтобы оплачивать стоимость приборного оснащения при запуске приложения. Если быстрый запуск имеет значение, то это могло бы пригодиться.

Другой вариант использования — расширение байт-кода сущностей Java Persistence API (JPA). Например, Hibernate уже делает это с агентом, отслеживающим, какие объекты были видоизменены (так называемая *грязная проверка*), не прибегая к проверке каждого поля. Делать это можно по ссылке. Как следствие, Hibernate уже предлагает плагины для инструментов сборки и IDE, которые делают это в процессе сборки.

Последний пример: действительно хорошим потенциальным плагином jlink будет тот, который индексирует аннотации во время соединения и делает данный индекс доступным во время выполнения. Это может значительно сократить время запуска приложений, сканирующих путь модуля для аннотированных компонентов и объектов. Фактически учебник по плагину, который я дал в сноске, делает именно это.

Использование плагинов jlink

> **Определение: параметр командной строки для плагина: --${имя}**
>
> Не вдаваясь в теорию, используем некоторые из представленных плагинов. Но как? Это довольно просто: `jlink` *автоматически создает параметр командной строки* `--${`*имя*`}` на основе имени каждого плагина. Способ передачи дальнейших параметров зависит от плагина и описан в `jlink --list-plugins`.

Удаление символов отладки — хороший способ уменьшить размер образа. Для этого создайте образ с помощью `--strip-debug`:

```
$ jlink
    --module-path ${jdk-9}/jmods
    --add-modules java.base
    --strip-debug
    --output jdk-base-stripped
```

Вот и все: `lib/modules` просто сократился с 23 Мбайт только для базового модуля до 18 Мбайт (в Linux).

Упорядочение содержимого `lib/modules` путем размещения более важных файлов первыми может сократить время запуска (хотя я сомневаюсь, что эффект будет заметен):

```
$ jlink
    --module-path ${jdk-9}/jmods
    --add-modules java.base
    --order-resources=**/module-info.class,/java.base/java/lang/**
    --output jdk-base-ordered
```

Таким образом, сначала идут дескрипторы модулей, за которыми следуют классы из пакета `java.lang`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Теперь, когда вы знаете, как использовать плагины, пришло время протестировать парочку из них. Мы сделаем это в двух разделах, первый из которых посвящен уменьшению размера (см. подраздел 14.4.2), а второй — повышению производительности (см. подраздел 14.4.3). Поскольку данная функция является развивающейся, а также довольно специализированной, я не буду вдаваться в подробности — официальная документация по `jlink` и `jlink --list-plugins`, хотя и содержит гораздо меньше слов, гораздо более точно показывает, как их применять.

14.4.2. Уменьшение размера образа

Рассмотрим плагины, уменьшающие размер образа, один за другим и измерим, как далеко они нас заведут. Я хотел бы протестировать их на образе приложения, но *ServiceMonitor* имеет всего около десятка классов, так что это бессмысленно. И я не смог найти реальное приложение, которое свободно доступно и полностью модульно, включая его зависимости (в образах нет автоматических модулей, помните?). Вместо этого измерим эффекты на трех разных образах среды выполнения (неизмененный размер в скобках):

- *base* — просто *java.base* (45 Мбайт);
- *services* — *java.base* плюс все поставщики сервисов (150 Мбайт);
- *java* — все модули *java.** и *javafx.**, но без поставщиков сервисов (221 Мбайт).

Интересно, что больший размер *java*, по сравнению с *services*, появляется не из-за количества байт-кода (`lib/modules` немного меньше в *java* по сравнению с *services*), а из-за собственных библиотек, в частности из кода WebKit, связанного для JavaFX `WebView`. Это поможет вам понять поведение плагинов, когда вы будете работать над уменьшением размера образа. (Кстати, я уменьшаю размер образа в Linux, но пропорции должны быть одинаковыми в других ОС.)

Сжатие образа

Определение: плагин сжатия

Плагин *сжатия* предназначен для уменьшения размера `lib/modules`. Он управляется параметром `--compress=${`*значение*`}`, который имеет три возможных значения:

- `0` — без сжатия (по умолчанию);
- `1` — дублировать и совместно использовать строковые литералы (что означает `"text"` в `String s = "text";`);
- `2` — Zip-сжатие `lib/modules`.

Необязательный список шаблонов может быть включен с помощью `--compress=${`*значение*`}:filter=${`*список_паттернов*`}` в этом случае сжимаются только файлы, которые соответствуют шаблонам.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Эта команда создает сжатый образ среды выполнения только с базовым модулем:

```
$ jlink
    --module-path ${jdk-9}/jmods
    --add-modules java.base
    --output jdk-base
    --compress=2
```

Очевидно, что не нужно пробовать `0`. Для `1` и `2` я получил следующие результаты:

- *base* — 45 Мбайт -> 39 Мбайт (`1`) -> 33 Мбайт (`2`);
- *services* — 150 Мбайт -> 119 Мбайт (`1`) -> 91 Мбайт (`2`);
- *java* — 221 Мбайт -> 189 Мбайт (`1`) -> 164 Мбайт (`2`).

Как видите, степень сжатия неодинакова для всех образов. Размер образа *services* можно уменьшить почти на 40 %, но более крупный образ *java* — лишь на 25 %. Это потому, что плагин сжатия работает только с библиотеками `lib/modules`, но, как мы уже обсуждали, они имеют практически одинаковый размер в обоих образах. Соответственно, уменьшение абсолютного размера аналогично: около 60 Мбайт для обоих изображений, что составляет более 50 % от исходного размера `lib/modules`.

ПРИМЕЧАНИЕ

Сжатие Zip с параметром --compress=2 увеличит время запуска — обычно оно становится тем больше, чем больше образ. Обязательно замерьте время, если это важно для вас.

Исключение файлов и ресурсов

Определение: плагины exclude-files и exclude-resources

Плагины *exclude-files* и *exclude-resources* позволяют исключать файлы из окончательного образа. Соответствующие параметры `--exclude-files=${`*`список_паттернов`*`}` и `--exclude-resources=${`*`список_паттернов`*`}` принимают список паттернов, с которыми сопоставляются файлы, подлежащие исключению.

Как я упоминал при сравнении начальных размеров образов *services* и *base*, за это отвечает в основном собственный двоичный файл для JavaFX `WebView`, который делает *java* больше. На моей машине это файл 73 Мбайт `lib/libjfxwebkit.so`. Вот как его можно исключить с помощью `--exclude-files`:

```
$ jlink
    --module-path ${jdk-9}/jmods
    --add-modules java.base
    --output jdk-base
    --exclude-files=**/libjfxwebkit.so
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Полученный образ на 73 Мбайт меньше. Два предостережения:

- тот же эффект будет и при удалении файла из образа вручную;
- уменьшение образа делает модуль *javafx.scene.web*, который в основном содержит только `WebView`, практически бесполезным, поэтому, вероятно, лучше просто не добавлять данный модуль.

Помимо экспериментов и обучения, плохая практика — исключать контент, который поставляется с платформенными модулями. Обязательно тщательно изучите любое решение, так как оно может повлиять на стабильность JVM.

Гораздо лучше использовать эти плагины, чтобы исключить файлы, содержащие приложение или JAR-файлы зависимостей, которые не нужны в образе приложения. Это могут быть документация, нежелательные исходные файлы, собственные двоичные файлы для ОС, ненужные вам, конфигурации или любые другие многочисленные элементы, которые гениальные разработчики поместили в свои архивы. Бессмысленно также сравнивать уменьшение размера: вы сэкономите место, занимаемое исключенными файлами.

Исключение ненужных языковых стандартов

Языковые стандарты — нечто из платформенных модулей, которые можно попробовать удалить. Как вы обнаружили в подразделе 14.1.3, базовый модуль может работать лишь с английскими языковыми стандартами, тогда как *jdk.localedata* содержит информацию обо всех других стандартах, поддерживаемых Java. К сожалению, все эти другие, вместе взятые, имеют размер около 16 Мбайт. Это немного излишне, если вам нужен только один или даже несколько неанглийских языков.

> **Определение: плагин include-locales**
>
> Вот где в игру вступает плагин *include-locales*. Он используется как `--include-locales=${`*`языки`*`}`, где `${`*`языки`*`}` — разделенный запятыми список языковых тегов BCP 47 (которые выглядят как `en-US`, `zh-Hans` и `fi-FI`), и полученный образ будет содержать только эти языки.
>
> Плагин работает лишь в том случае, если модуль *jdk.localedata* превращает его в образ, поэтому не *включает* столько языковых стандартов, сколько содержит базовый модуль, так как *исключает* все другие языковые стандарты из *jdk.localedata*.

В листинге 14.4 создан образ приложения для *ServiceMonitor*, который включает в себя все файлы *jdk.localedata*, поскольку приложение задействует финский формат для вывода. Это увеличило размер изображения на 16 Мбайт, и теперь вы знаете, как его уменьшить. В листинге 14.7 для этого используется `--include-locales=fi-FI`. Полученный образ немногим больше, чем без *jdk.localedata* (если быть точным, то 168 Кбайт). Отлично!

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Листинг 14.7. Создание образа приложения ServiceMonitor с данными о финском языке

```
$ jlink
    --module-path ${jdk-9}/jmods:mods
    --add-modules monitor,
        monitor.observer.alpha,monitor.observer.beta,
        jdk.localedata
    --output jdk-monitor
    --include-locales=fi-FI
```

Платформенный модуль для языковых стандартов необходимо добавить в образ — либо явно (как здесь), либо неявно (по требованию или с помощью --bind-services)

Все языковые стандарты, кроме fi-FI (финский), удалены из jdk.localedata

Уменьшение размера образа за счет исключения языковых стандартов зависит от того, сколько стандартов вам нужно. Если вы разрабатываете интернационализированное приложение для глобальной аудитории, то не получится сэкономить много, но я думаю, что это необычный случай. Если приложение поддерживает лишь несколько языков или даже дюжину, исключая все другие, то вы сэкономите почти все 16 Мбайт. Стоит ли это усилий — решать только вам.

Удаление отладочной информации

При отладке Java-кода в IDE вы обычно видите красиво отформатированный, именованный и даже закомментированный исходный код. Это связано с тем, что среда извлекает фактические источники, принадлежащие данному коду, связывает их с исполняемым в настоящее время байт-кодом и удобно отображает их. Это лучший сценарий.

Если источников нет, то вы все равно можете получить читабельный код, если в дополнение к именам параметров полей и методов (которые всегда присутствуют в байт-коде) видите правильные имена переменных (необязательно присутствующие). Это происходит, когда декомпилированный код содержит *символы отладки*. Данная информация значительно упрощает отладку, но, конечно, занимает много места. И `jlink` позволяет вычеркнуть данные символы.

> **Определение: плагин strip-debug**
>
> Если плагин *strip-debug* `jlink` активирован с помощью `--strip-debug`, то удалит все символы отладки из байт-кода образа, тем самым уменьшив размер файла `lib/modules`. У этой опции нет других параметров.

Я использовал `--strip-debug` в подразделе 14.4.1, так что избавлю вас от повторения. Посмотрим, как это уменьшает размеры образа:

- *base* — 45 Мбайт -> 40 Мбайт;
- *services* — 150 Мбайт -> 130 Мбайт;
- *java* — 221 Мбайт -> 200 Мбайт.

Это примерно 10 % от общего размера образа, но помните, что вышеизложенное касается только `lib/modules`, которая уменьшается примерно на 20 %.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

Предупреждение: отладка кода без исходников и без отладочных символов — адская задача. Если вы время от времени используете удаленную отладку для подключения к работающему приложению и анализируете происходящее, то не будете счастливы, убрав эти символы отладки, а те несколько мегабайт, которые вы сохранили, не будут действительно важны. Тщательно продумайте --strip-debug!

Собираем все вместе

Хотя исключение файлов и ресурсов лучше оставить для модулей приложения, другие параметры хорошо работают на чистых образах среды выполнения. Соединим их все и попробуем создать наименьший возможный образ для трех выбранных модулей. Ниже представлена команда только для *java.base*:

```
$ jlink
    --module-path ${jdk-9}/jmods
    --add-modules java.base
    --output jdk-base
    --compress=2
    --strip-debug
```

Результаты таковы:

- *base* — 45 Мбайт -> 31 Мбайт;
- *services* — 150 Мбайт -> 75 Мбайт (я также удалил все языковые стандарты, кроме `fi-FI`);
- *java* — 221 Мбайт -> 155 Мбайт (или 82 Мбайт, во вред JavaFX `WebKit`).

Не так уж и плохо, а?

14.4.3. Улучшение производительности во время выполнения

Как вы уже видели, существует несколько способов уменьшить размер образа приложения или выполнения. Однако я предполагаю, что большинство разработчиков с нетерпением ждут улучшения производительности, особенно после того, как Specter и Meltdown отняли у них часть циклов ЦП.

ВАЖНАЯ ИНФОРМАЦИЯ

К сожалению, у меня не так много хороших новостей в этом отношении: оптимизация производительности с помощью jlink все еще находится на ранних стадиях, и большинство из существующих или предполагаемых оптимизаций фокусируется на улучшении времени запуска, а не на долгосрочной производительности во время выполнения.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Один из существующих плагинов, включенных по умолчанию, — *system-modules*, который предварительно вычисляет граф системных модулей и сохраняет его для быстрого доступа. Таким образом, JVM не нужно анализировать и обрабатывать декларации модулей, проверяя надежную конфигурацию для каждого запуска.

Другой плагин, *class-for-name*, заменяет байт-код, такой как `Class.forName("some.Type")` на `some.Type.class`, поэтому можно избежать сравнительно дорогого поиска класса по имени. Мы кратко рассмотрели *order-resources*, и добавить тут особо нечего.

Единственный другой плагин, связанный с производительностью, который в настоящее время поддерживается, — *генерирующий generate-jli-classes*. При правильной настройке он может переносить затраты на инициализацию лямбда-выражений со времени выполнения на время компоновки. Но изучать то, как это сделать, можно, лишь хорошо понимая дескрипторы методов, вследствие чего я не буду здесь останавливаться на этом.

И это все, что касается улучшения производительности. Я понимаю, если вы разочарованы отсутствием больших успехов в данной области, но позвольте мне отметить, что JVM уже довольно оптимизирована. Все низко висящие плоды (и многие выше по дереву) уже собраны, и потребуется немного изобретательности, времени и смекалки, чтобы достать другие. Инструмент `jlink` все еще молод, и я уверен: команда разработчиков JDK и сообщество своевременно его используют.

Обмен данными классов приложения в Java 10

Оптимизация, представленная Java 10, не связана напрямую с `jlink`: совместное использование данных классов приложения[1]. Эксперименты показывают, что она может привести к запуску приложения, который происходит на 10–50 % быстрее. Что интересно, эта техника, примененная в образе приложения, позволяет создать еще более оптимизированный модуль развертывания.

14.5. Параметры для jlink

В целях вашего удобства в табл. 14.2 перечислены все параметры командной строки для `jlink`, которые обсуждаются в этой книге. Узнать больше можно в официальной документации по адресу https://docs.oracle.com/javase/9/tools/jlink.htm или с помощью вызова `jlink --help` и `jlink --list-plugins`.

[1] Для получения дополнительной информации см. мой пост в блоге «Улучшение времени запуска Java 10 с помощью обмена данными классов приложения», blog.codefx.org/java/application-class-data-sharing.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Таблица 14.2. Алфавитная таблица выбранных параметров jlink, включая плагины. Описания основаны на документации, а ссылки указывают на подразделы в этой книге, в которых подробно объясняется, как использовать параметры

Параметр	Описание	Ссылка
--add-modules	Определяет корневые модули для включения в образ	14.1.1
--bind-services	Добавляет все поставщики сервисов, которые используют разрешенные модули	14.1.3
--class-for-name	Заменяет Class::forName статическим доступом (плагин)	14.4.3
--compress, -c	Делит строковые литералы и сжимает lib/modules (плагин)	14.4.2
--exclude-files, --exclude-resources	Исключает указанные файлы и ресурсы (плагин)	14.4.2
--generate-jli-classes	Прегенерирует дескрипторы метода (плагин)	14.4.3
--include-locales	Удаляет все, кроме указанных языковых стандартов из jdk.localedata (плагин)	14.4.2
--launcher	Создает собственный сценарий запуска для приложения в bin	14.2.2
--list-plugins	Перечисляет доступные плагины	14.4.1
--module-path, -p	Указывает, где найти платформенные модули и модули приложения	14.1.1
--order-resources	Упорядочивает ресурсы в lib/modules (плагин)	14.4.1
--output	Создает образ в указанном месте	14.1.1
--strip-debug	Удаляет символы отладки из байт-кода образа (плагин)	14.4.2
--suggest-providers	Перечисляет обозреваемых поставщиков для указанных сервисов	14.1.3

Резюме

- Инструмент командной строки `jlink` создает образы среды выполнения из выбранных платформенных модулей (используйте `jdeps`, чтобы определить, какие из них нужны приложению). Чтобы использовать `jlink`, приложение должно работать на Java 9+, но быть модульным ему не обязательно.
- После того как приложение и его зависимости станут полностью модульными (без использования автоматических модулей), `jlink` может создавать с его помощью образы приложений, включая модули приложения.
- Для всех вызовов `jlink` необходимо указать следующее:
  - где найти модули (включая платформенные), с `--module-path`;
  - корневые модули для разрешения, с параметром `--add-modules`;
  - выходной каталог для полученного образа, с `--output`.
- Имейте в виду, как `jlink` разрешает модули:
  - сервисы не связаны по умолчанию;
  - необязательные зависимости с использованием статических не разрешаются;

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

- автоматические модули не допускаются;
- обязательно добавьте требуемые поставщики сервисов или дополнительные зависимости по отдельности с помощью `--add-modules` или свяжите все поставщики с помощью `--bind-services`.

- Следите за сервисами платформы, от которых можете неявно зависеть, даже не осознавая этого. Одни из кандидатов — `charsets` (`jdk.charsets`), `locales` (`jdk.localedata`), файловая система Zip (`jdk.zipfs`) и поставщики безопасности (различные модули).
- Образ выполнения, сгенерированный `jlink`:
  - связан с ОС, для которой были построены платформенные модули, выбранные с помощью `--module-path`;
  - имеет ту же структуру каталогов, что и JDK, и JRE;
  - объединяет платформенные модули и модули приложения (вместе известные как системные модули) в `lib/modules`;
  - содержит только двоичные файлы (в `bin`), для которых были включены необходимые модули.
- Чтобы запустить образ приложения, используйте либо `bin/java --module ${`*`начальный_модуль`*`}` (путь к модулю не требуется, поскольку системные модули автоматически разрешаются), либо средство запуска, созданное с помощью `--launcher ${`*`имя`*`}=${`*`модуль`*`}/${`*`основной_класс`*`}`.
- С помощью образов приложений путь модуля можно использовать для добавления дополнительных модулей (особенно тех, которые предоставляют сервисы). Модули, чьи имена совпадают с системными, игнорируются.
- Тщательно оцените последствия для безопасности, производительности и стабильности предоставления образов приложений, когда не можете легко заменить их более новыми версиями.
- Различные параметры `jlink`, которые активируют плагины, предлагают способы уменьшения размеров образа (например, `--compress`, `--exclude-files`, `--exclude-resource`, `--include-locales` и `--strip-debug`) или улучшить производительность (в основном время запуска: `--class-for-name`, `--generate-jli-classes` и `--order-resources`). В будущем можно ожидать большего; эта область все еще находится на стадии разработки.
- API плагинов `jlink` еще не стандартизирован, чтобы облегчить его эволюцию на этом раннем этапе; как следствие, разработка и использование сторонних плагинов затруднены.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

15 Собираем все вместе

В этой главе

- Навороченная версия ServiceMonitor.
- Когда использовать модули.
- Как может выглядеть идеальный модуль.
- Поддержание чистоты модульных деклараций.
- Сравнение системы модулей для создания инструментов, OSGi и микросервисов.

Теперь, когда мы рассмотрели почти все, что нужно знать о системе модулей, пришло время подвести итоги. В этой, последней главе я хочу расставить все точки над i и дать несколько советов по созданию классных модульных приложений.

Первый шаг — я покажу пример того, как можно объединить обсуждаемые в книге различные функции, применив большинство из них к приложению *ServiceMonitor* (см. раздел 15.1). Затем углублюсь в ряд более общих вопросов, которые помогут вам решить, следует ли создавать модули, к чему стремиться при этом и как тщательно развивать модульные декларации, чтобы они оставались безупречно чистыми (см. раздел 15.2). В завершение я приведу обзор технологического ландшафта системы модулей см. (раздел 15.3) и мое видение модульной экосистемы Java (см. раздел 15.4).

15.1. Добавляем навороты в ServiceMonitor

Глава 2 показала анатомию приложения *ServiceMonitor*. В разделе 2.2 вы создали простые модули, в которых использовались лишь простые директивы `requires` и `exports`. С тех пор мы не только подробно обсудили это, но и изучили более расширенные функции системы модулей. Мы рассмотрели каждую из них отдельно, но теперь я хочу их собрать вместе.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Чтобы насладиться приложением *ServiceMonitor* во всей его красе, ознакомьтесь с веткой `features-combined` репозитория. Листинг 15.1 содержит декларации для всех модулей в *ServiceMonitor*.

Листинг 15.1. Приложение ServiceMonitor с расширенными функциями, представленными в книге

```
module monitor.observer {
    exports monitor.observer;
    exports monitor.observer.utils
        to monitor.observer.alpha, monitor.observer.beta;
}

module monitor.observer.alpha {
    requires monitor.observer;
    provides monitor.observer.ServiceObserverFactory
        with monitor.observer.alpha.AlphaServiceObserverFactory;
}

// [...]

module monitor.statistics {
    requires transitive monitor.observer;
    requires static stats.fancy;
    exports monitor.statistics;
}

module stats.fancy {
    exports stats.fancy;
}

module monitor.persistence {
    requires transitive monitor.statistics;
    requires hibernate.jpa;
    exports monitor.persistence;
    opens monitor.persistence.entity;
}

module monitor.rest {
    requires transitive monitor.statistics;
    requires spark.core;
    exports monitor.rest;
}

module monitor {
    requires monitor.observer;
    requires monitor.statistics;
    requires monitor.persistence;
    requires monitor.rest;
    uses monitor.observer.ServiceObserverFactory;
}
```

monitor.observer.utils предназначен для реализаций обозревателей, поэтому экспортируется только в некоторые из них (см. подраздел 15.1.2)

Потребитель (monitor) и реализации (например, monitor.observer.alpha) API обозревателя отделены с помощью сервисов (см. подраздел 15.1.3)

monitor.observer.beta и monitor.observer.gamma здесь не отображаются; они выглядят так же, как monitor.observer.alpha

Некоторые модули представляют типы из другого модуля в своем API и не могут использоваться без этого модуля, поэтому подразумевают его читабельность (см. подраздел 15.1.1)

stats.fancy отсутствует в каждом развертывании, а monitor.statistics рефлексирует его, помечая свою зависимость от этого модуля как необязательную (см. подраздел 15.1.1)

Некоторые модули представляют типы из другого модуля в своем API и не могут использоваться без этого модуля, поэтому подразумевают его читабельность (см. подраздел 15.1.1)

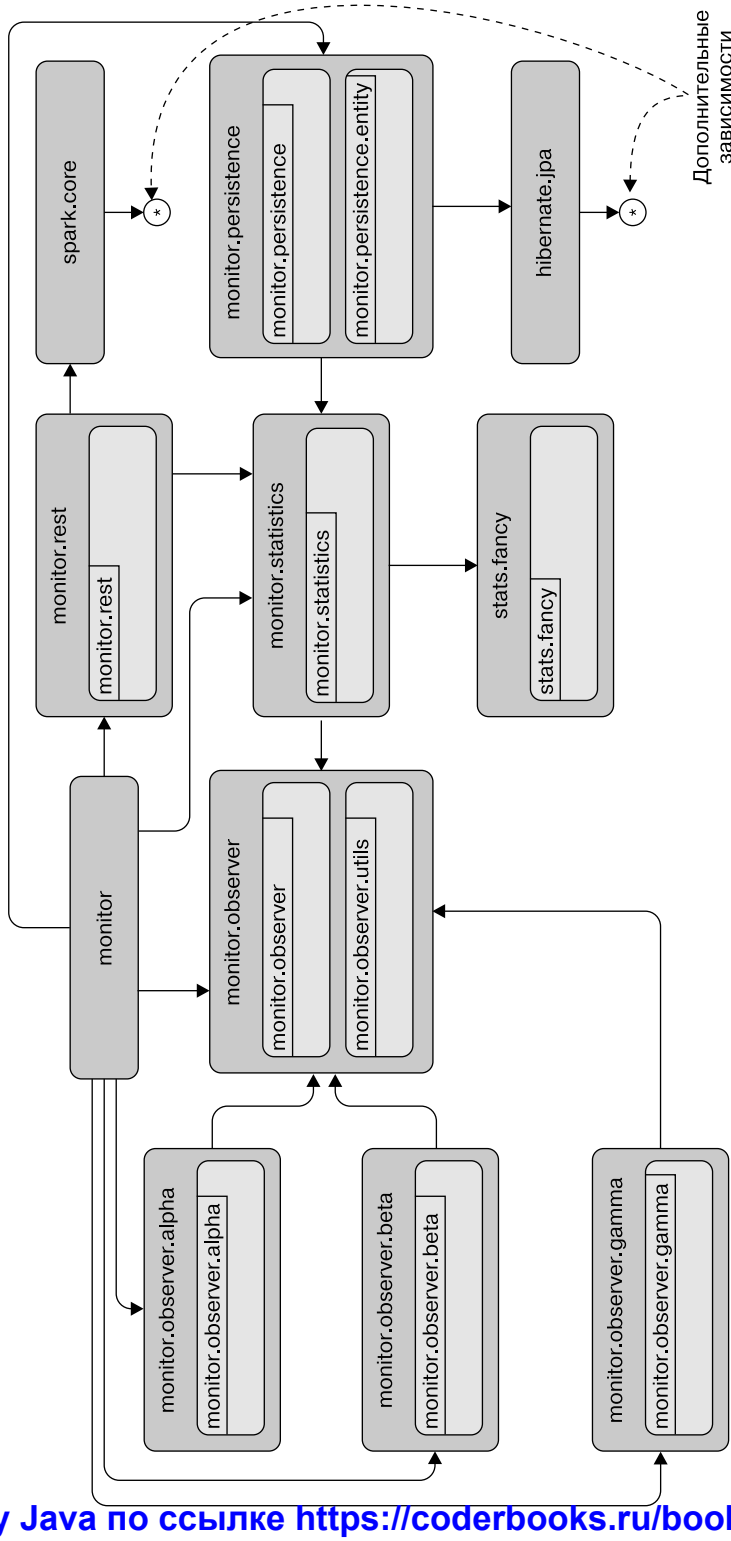
monitor.persistence открывает пакет, содержащий его объекты состояния для рефлексии (см. подраздел 15.1.2)

Ни версия Hibernate, ни версия Spark, которую использует ServiceMonitor, не являются модуляризованными, поэтому hibernate.jpa и spark.core не автоматические модули (см. подраздел 15.1.5)

Потребитель (monitor) и реализации (например, monitor.observer.alpha) API обозревателя отделены с помощью сервисов (см. подраздел 15.1.3)

Если сравнить этот листинг с листингом 2.2 или посмотреть на рис. 15.1, можно увидеть, что фундаментальная структура *ServiceMonitor* осталась почти неизменной. Но, присмотревшись, можно увидеть ряд улучшений. Пройдемся по ним по порядку.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/



Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Рис. 15.1. Сравнение модульных графов для приложения ServiceMonitor в зависимости от использованных функций. Первый вариант использует только простые директивы exports и requires *(сверху)*, тогда как второй полностью задействует уточненные зависимости и экспорты, а также сервисы *(снизу)*. (Базовый вариант был расширен и включает те же модули и пакеты, что и расширенный)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

15.1.1. Разнообразные зависимости

Одни из легко обнаруживаемых изменений — директивы `requires transitive` и `requires optional`. Хотя даже обычные директивы `requires` в большинстве случаев — правильный выбор, значительная часть зависимостей немного сложнее.

Наиболее очевидный случай — необязательные зависимости, когда модуль использует типы из другого модуля и, следовательно, должен быть скомпилирован с ним, но зависимость может все еще отсутствовать во время выполнения. Это в точности относится к *monitor.statistics* и *stats.fancy*, вследствие чего зависимость устанавливается с помощью директивы `requires static`.

Затем система модулей обеспечит наличие *stats.fancy* при компиляции *monitor.statistics* (что имеет смысл, поскольку в противном случае компиляция не удалась бы) и добавит ребро чтения из *monitor.statistics* в *stats.fancy*, если последний попадет в модульный граф (и это имеет смысл, ведь в противном случае *monitor.statistics* не сможет получить доступ к типам из *stats.fancy*). Но *stats.fancy* может не попасть в граф, и в этом случае *monitor.statistics* должен обработать его отсутствие.

Листинг 15.2. Проверка наличия дополнительной зависимости stats.fancy

```
private static boolean checkFancyStats() {
    boolean isFancyAvailable = isModulePresent("stats.fancy");
    String message = "Module 'stats.fancy' is"
            + (isFancyAvailable ? " " : " not ")
            + "available.";
    System.out.println(message);
    return isFancyAvailable;
}

private static boolean isModulePresent(String moduleName) {
    return Statistician.class
        .getModule()
        .getLayer()
        .findModule(moduleName)
        .isPresent();
}
```

Необязательные зависимости подробно обсуждаются в разделе 11.2.

Другой случай немного менее очевиден, чем необязательные зависимости, но не менее распространен — возможно, даже более. Например, модуль *monitor.rest* предоставляет этот метод в своем публичном API:

```
public static MonitorServer create(Supplier<Statistics> statistics) {
    return new MonitorServer(statistics);
}
```

Но `Statistics` поступает из *monitor.statistics*, поэтому любой модуль, использующий *rest*, должен читать *Statistics* или не может получить доступ к `Statistics` и, следовательно, не может создать `MonitorServer`. Другими словами, *rest* бесполезен для модулей, также не читают *statistics*. В приложении *ServiceMonitor* это происходит на

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

удивление часто: каждый модуль, которому требуется хотя бы один другой модуль и который экспортирует пакет, оказывается в такой ситуации.

Здесь это происходит значительно чаще, чем в реальной жизни, и случается так лишь вследствие настолько малого размера модулей, что почти весь их код является открытым API — было бы удивительно, если бы они постоянно не выставляли типы своих зависимостей в собственных API. Таким образом, хоть на практике вышеописанное и происходит реже, вы все равно можете ожидать, что будете видеть это ежедневно — в JDK выставлено примерно 20 % зависимостей.

Чтобы пользователи не догадывались о том, какие другие модули им нужны явно (это громоздко и приводит к переполнению деклараций модулей), система модулей предлагает `requires transitive`. Поскольку *rest* `requires transitive` *statistics*, любой модуль, читающий *rest*, читает и *statistics*, и, таким образом, пользователи *rest* избавлены от догадок. Подразумеваемая читабельность подробно обсуждается в разделе 11.1.

15.1.2. Снижение видимости

Другое отличие от первоначальных версий приложения, представленных в разделе 2.2, состоит в том, что его модули усерднее работают над тем, чтобы уменьшить свою поверхность API. Обновленные модули используют значительно меньше простых директив `exports`:

- благодаря сервисам обозреватели больше не должны экспортировать свои реализации;
- с помощью квалифицированных экспортов пакет `monitor.observer.utils` в *monitor.observer* доступен только для выбранного набора модулей;
- *monitor.persistence* открывает пакет сущностей вместо экспорта, что делает его доступным лишь во время выполнения.

Эти изменения уменьшают объем кода, к которому легче получить доступ из любого случайного модуля, то есть разработчики могут изменять больше кода внутри модуля, не беспокоясь о последствиях для последующих потребителей. Таким образом, идет сокращение поверхности API — что не только хорошо для удобства сопровождения фреймворков и библиотек, но может принести пользу и крупным приложениям с большим количеством модулей. В разделе 11.3 представлен квалифицированный экспорт, а в разделе 12.2 — открытые пакеты.

15.1.3. Отделение от сервисов

Единственное структурное изменение модульного графа (по сравнению с разделом 2.2) заключается в том, что *monitor* больше не зависит напрямую от реализаций обозревателя. Вместо этого он зависит только от модуля, предоставляющего API, *monitor.observer*, и использует `ServiceObserverFactory` в качестве сервиса. Все три реализующих модуля обеспечивают данную услугу конкретными реализациями, и система модулей объединяет эти две стороны.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Это гораздо больше, чем просто эстетическое улучшение. Сервисы позволяют настроить аспекты поведения приложения — какие сервисы оно может обозревать — во время запуска. Новые реализации можно добавить, а устаревшие — удалить, добавляя или удаляя модули, которые предоставляют этот сервис, — изменений модуля *monitor* не требуется, и, следовательно, одни и те же артефакты можно использовать, не перестраивая их. Чтобы узнать все о сервисах, ознакомьтесь с главой 10.

15.1.4. Загрузка кода со слоями во время выполнения

Сервисы позволяют нам определять поведение приложения во время запуска, однако мы пошли еще дальше. Этого не видно в объявлениях модуля, но, позволив модулю *monitor* создавать новые слои, мы разрешили приложению запускать сервисы обозревателя во время выполнения, для которого у него даже не было реализации `ServiceObserver` при запуске. По требованию *monitor* создаст новый модульный граф и вместе с новым загрузчиком классов загрузит дополнительные классы и обновит свой список обозревателей (листинг 15.3).

Листинг 15.3. Создание нового слоя с графом, написанным для модулей в этих путях

```
private static ModuleLayer createLayer(Path[] modulePaths) {
    Configuration configuration = createConfiguration(modulePaths);
    ClassLoader thisLoader = getThisLoader();
    return getThisLayer()
        .defineModulesWithOneLoader(configuration, thisLoader);
}

private static Configuration createConfiguration(Path[] modulePaths) {
    return getThisLayer()
        .configuration()
        .resolveAndBind(
            ModuleFinder.of(),
            ModuleFinder.of(modulePaths),
            Collections.emptyList()
        );
}
```

Такое поведение особенно интересно для приложений, которые развертываются нечасто и где перезапуски неудобны. На ум приходят сложные приложения для ПК, но веб-сервер, который работает на территории клиента и должен быть приемлемо настраиваемым, тоже может подойти. Описание того, что такое слои и как их создавать, см. в разделе 12.4.

15.1.5. Обработка зависимостей от простых JAR-файлов

Еще одна деталь, неочевидная из объявлений модуля, — это статус модуляризации сторонней зависимости *ServiceMonitor*. Ни версия Hibernate, ни версия Spark, которую оно использует, еще не являются модульными, и они по-прежнему поставля-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ются в виде простых файлов JAR. Поскольку явные модули запрашивают их, они должны находиться в пути модуля, где система модулей превращает простые JAR-файлы в автоматические модули.

Таким образом, хотя *ServiceMonitor* полностью модульное, оно тем не менее может зависеть от немодульных JAR-файлов. Если посмотреть на всю картину с точки зрения экосистемы, где модули JDK находятся внизу, а модули приложений — вверху, то это фактически нисходящие усилия по модуляризации.

В частности, автоматические модули описаны в разделе 8.3, но здесь применяется весь материал главы 8. Если вы хотите узнать о стратегиях модуляризации, то ознакомьтесь с разделом 9.2.

15.2. Советы для разработчиков модульных приложений

На протяжении всей книги мы потратили много времени на изучение того, как использовать различные инструменты системы модулей для решения отдельных задач. Это, безусловно, самая важная задача книги о JPMS, но я не отпущу вас, не представив хотя бы краткий перечень всего набора инструментов.

Первый вопрос: хотите ли вы использовать эти инструменты? Если без намеков, то хотите ли вы создавать модули (см. подраздел 15.2.1)? Как только это будет решено, мы попытаемся определить, как может выглядеть идеальный модуль (см. подраздел 15.2.2). Затем сосредоточимся на том, как сохранить объявления модулей в прекрасной форме (см. подраздел 15.2.3) и какие изменения могут нарушить код ваших пользователей (см. подраздел 15.2.4).

15.2.1. Модуль или нет?

После всего, что вы узнали о системе модулей: ее особенностях, недостатках, обещаниях и ограничениях — возможно, вы все еще спрашиваете себя, следует ли модуляризовывать ваши JAR-файлы. В конце концов, только вы и ваша команда можете ответить на этот вопрос для конкретного проекта, но я могу высказать свое мнение по данной теме.

Как я уже говорил, я убежден, что система модулей предлагает множество преимуществ, которые важны для библиотек, фреймворков и большинства нетривиальных приложений. Для меня особенно выделяются строгая инкапсуляция, разделение с помощью сервисов (это можно сделать и без модулей, хотя и менее комфортно) и образы приложений.

Что тем не менее мне нравится больше всего — сами декларации модулей: они всегда являются истинным представлением архитектуры проекта и обеспечат значительные преимущества для каждого разработчика и архитектора, который работает над этими аспектами своей системы, улучшая тем самым ее общее удобство сопровождения. (Я углубляюсь в данную тему в подразделе 15.2.3.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

По этим причинам я по умолчанию запускаю каждый новый проект, разработанный на Java 9+, с модулями. (Теоретически причины, специфичные для проекта, могут убедить меня в обратном, но я не могу придумать ничего подобного.) Если зависимости начинают создавать слишком много проблем, когда помещаются в путь модуля (например, могут разделять пакеты, см. раздел 7.2), то довольно легко выйти из системы модулей, используя путь класса вместо пути модуля. При работе с модулями с самого начала их создание и развитие, по сути, займет почти совсем немного времени, в то время как улучшенное удобство сопровождения значительно сократит количество путаницы, которую необходимо разрешить по мере роста и старения проекта.

Если вы не уверены, то сначала попробуйте. Создайте демонстрационный проект с модулями или, что еще лучше, небольшим приложением с реальными пользователями и требованиями. Некритические, фирменные инструменты создают отличных подопытных кроликов.

Когда дело доходит до модуляризации существующих проектов, ответ в гораздо большей степени «зависит от обстоятельств». Объем работы, который необходимо сделать, куда более очевиден, но выгоды столь же ощутимы. Фактически чем больше работы нужно выполнить, тем лучше будет результат. Подумайте об этом: какие приложения сложнее всего модуляризовать? Состоящие из большего количества артефактов, более запутанные и менее удобные в сопровождении. Но это и те приложения, которые больше всего выиграют от изучения и работы над их структурой. Поэтому будьте осторожны, когда кто-то предполагает, что модуляризация существующего проекта имеет низкие затраты и большие выгоды (или наоборот).

ВАЖНАЯ ИНФОРМАЦИЯ

В конечном итоге ожидаемый оставшийся срок службы проекта может привести к разрыву связей. Чем дольше необходимо сопровождать проект, тем ниже относительные затраты и тем выше преимущества модуляризации. Другими словами, чем длиннее оставшееся время жизни, тем больше смысла имеет модуляризация.

Если вы работаете над проектом, в котором есть пользователи вне вашей команды, например с библиотекой или фреймворком, то следует также учитывать их потребности. Пусть модульность и не кажется вам стоящей, из нее можно будет извлечь значительную выгоду.

15.2.2. Идеальный модуль

Предположим, вы приняли решение и выбрали модули. Каков он — идеальный модуль? К чему вы стремитесь, когда кромсаете модули и пишете декларации? Еще раз: нет единого ответа, но есть несколько сигналов, на которые стоит настроить радар:

- размер модуля;
- поверхность API;
- связь между модулями.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Прежде чем по очереди обсудить каждый из них, я хочу добавить: даже если у вас есть представление об идеальном модуле, маловероятно, что вы будете выпускать один модуль за другим. В частности, если начнете с модуляризации существующего проекта, то, скорее всего, создадите парочку уродливых модулей.

Если вы работаете с приложением, то не нужно об этом беспокоиться — можно легко реорганизовать модули по мере необходимости. Для разработчиков библиотек и фреймворков жизнь усложняется. Как покажет подраздел 15.2.4, многие этапы рефакторинга могут нарушить код пользователей, поэтому у вас гораздо меньше свободы для развития.

Теперь обратимся к трем сигналам, которые можно отслеживать для оценки качества модуля: размер, поверхность и связь.

Держите свои модули маленькими

Модульные декларации предоставляют отличный инструмент для анализа и формирования границ между модулями, но они относительно слепы по отношению к тому, что происходит *внутри* модуля. Пакеты имеют циклические зависимости? Все классы и участники являются публичными? Это большой комок грязи? Все перечисленное может повредить во время разработки, но модульные декларации даже не отобразят этого.

Это значит, что чем больше модульных деклараций в проекте, тем лучше разработчики понимают и контролируют структуру кода (рис. 15.2). С другой стороны, между модулями, JAR и (как правило) проектами инструментов сборки существует взаимно однозначное отношение, поэтому большее количество деклараций модулей означает увеличение затрат на сопровождение и увеличение времени сборки. Это явный компромисс.

Тем не менее, как общее практическое правило, предпочитайте мéньшие модули бóльшим. Как только количество строк кода модуля становится пятизначным, стоит подумать о том, чтобы разделить его на части. Переход строк модуля в шесть знаков я рекомендую рассмотреть серьезно. Если же строки кода семизначны, то впереди серьезная работа по рефакторингу. (При возникновении проблем с нарушением циклических зависимостей между классами ознакомьтесь с подразделом 10.3.5, где для этого используются сервисы.)

ВАЖНАЯ ИНФОРМАЦИЯ

Учитывая все вышесказанное, не доверяйте никому, кто скажет, что для ваших модулей существует правильный размер, даже не взглянув на проект. Единственный правильный ответ на вопрос «Насколько маленькими или большими должны быть модули?» — это «Зависит от обстоятельств». Каждый модуль должен быть согласованным решением конкретной проблемы. Если у данной проблемы есть большое решение, то это нормально — не начинайте разбирать вещи, которые должны быть вместе.

Что должно быть объединено? Разрезая связующий модуль на две части, вы неизбежно получите довольно большую поверхность API между частями, что подводит нас к следующему аспекту, который мы должны обсудить.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

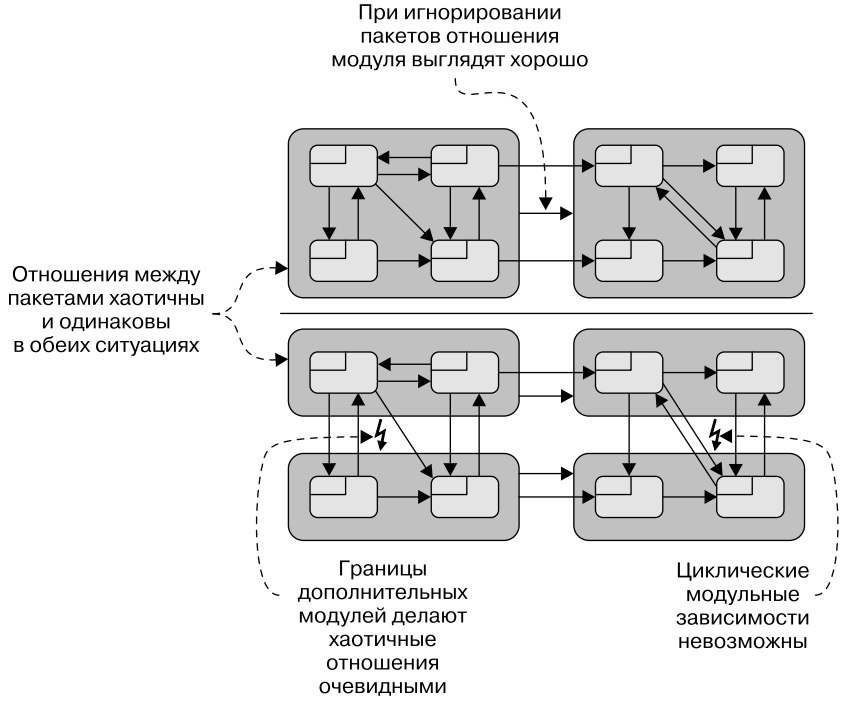


Рис. 15.2. Эти отношения пакетов, возможно, несколько хаотичны. Если вверху располагаются только два модуля, то это неочевидно. Лишь при попытке создать больше модулей *(снизу)* проблемы становятся очевидными. Границы дополнительного модуля обеспечили это понимание

Уменьшайте поверхности API

ВАЖНАЯ ИНФОРМАЦИЯ

Сила модулей в том, что они могут хранить свои внутренние устройства при себе. Это позволяет упростить рефакторинг внутри модуля и более тщательно развивать его общедоступный API. Учитывая данные преимущества, в целом более предпочтительным является меньшее количество простых директив exports. То же самое относится и к квалифицированному экспорту: чем меньше, тем лучше.

Как сравнить простой и квалифицированный экспорт? В рамках проекта нет большой разницы. Когда дело доходит до запутывания двух модулей, на самом деле не имеет значения, был ли экспорт квалифицирован. Тем не менее квалификация по крайней мере указывает, что API, возможно, не был разработан для общего использования, а это полезная информация, особенно в более крупных проектах.

Библиотеки и фреймворки, в отличие от приложений, всегда должны думать о том, как их экспорт влияет на проекты в зависимости от них. В этом сценарии квалифицированный экспорт в другие модули в том же проекте имеет ту же цену, как если бы пакет вообще не экспортировался, что, безусловно, является победой. Таким

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

образом, квалифицированный экспорт по-прежнему вносит вклад в поверхность API: почти столько же, сколько обычный экспорт в рамках проекта, но значительно меньше, чем экспорт за границы проекта.

Минимизируйте связи

Выберите два случайных фрагмента кода — не имеет значения, являются ли они методами, классами или модулями. При прочих равных условиях тот, у которого меньше зависимостей, более удобен в сопровождении. Причина проста: чем больше зависимостей, тем больше изменений могут разрушить связи.

Тем не менее данный аспект выходит за рамки простых зависимостей: в общем случае это вопрос взаимодействия. Если модуль не только зависит от другого, но и активно задействует все дюжины пакетов, которые экспортирует, то эти два модуля более тесно связаны. Это даже более верно в случае, когда квалифицированные экспорты — часть комплекса, поскольку они, по сути, говорят: «Это неправильно поддерживаемый API, но я все равно позволю вам его использовать».

ВАЖНАЯ ИНФОРМАЦИЯ

Описанное выходит за рамки отдельных модулей. Чтобы понять систему, нужно понимать не только части (здесь: модули), но и их связи (здесь: зависимости и связи). И если вы не будете осторожны, то система может заиметь гораздо больше соединений, чем частей (порядка количества модулей в квадрате (рис. 15.3)). Следовательно, слабосвязанные части — критически важный компонент для поддержания максимально простой системы.

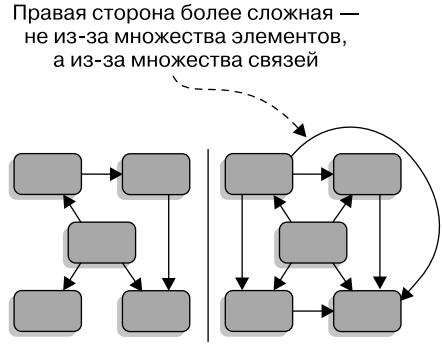


Рис. 15.3. Хотя оба графа имеют одинаковое количество узлов, они весьма различны по сложности. Тот, что слева, имеет столько же ребер, сколько узлов, тогда как тот, что справа, — около одного ребра на пару узлов. При добавлении нового узла левый граф получил бы одно или, возможно, два новых ребра, тогда как правый — около шести

Хороший способ отделить модули — это сервисы, как показано в главе 10. Они не только нарушают прямую зависимость между модулями, но и требуют наличия лишь одного типа, через который можно получить доступ ко всему API. Если только вы не превратите данный тип в кракена, подключающегося к десяткам других типов, это значительно уменьшит связь между модулями.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

Предупреждение: сервисы аккуратны, но их сложнее предсказать, чем простые зависимости. Вы не можете легко увидеть, как связаны две части кода, и не получите ошибок, если поставщики отсутствуют. Так что не переусердствуйте и не ставьте сервисы повсюду.

Вот что должно быть лакмусовой бумажкой: можете ли вы создать тип сервиса с достаточно маленьким API? Похоже, это можно использовать или предоставить более чем одним модулем с каждой стороны?

Если вы не уверены, то поищите ответ в пределах JDK. В официальной документации перечислены сервисы, которые использует или предоставляет модуль, и вы можете задействовать свою среду IDE для просмотра кодов пользователя и реализации.

Прослушайте декларацию модуля

ВАЖНАЯ ИНФОРМАЦИЯ

Секунду назад мы обсудили, что модули должны быть небольшими, иметь API-поверхность еще меньшего размера и быть слабо связанными с окружающей средой. В конце концов, эти советы можно свести к обманчиво простой формуле: поддерживать высокую связность и низкую связанность. Поиск размера модуля, количества exports и requires, а также сила каждой зависимости могут помочь в этом.

ПРИМЕЧАНИЕ

Эти показатели, как и любые другие, не должны стать самоцелью. Продуманная общая архитектура более важна, чем какие-то цифры. Хотя данная книга дает много инструментов и даже несколько советов для достижения цели, она не учит этому с нуля.

Кроме того, обратите внимание: три сигнала (размер, поверхность и связность) часто работают друг против друга. В качестве яркого примера возьмем приложение, состоящее лишь из одного модуля. У него, что очень вероятно, нет API; и только с одним артефактом не происходит особой связи. С другой стороны, кодовая база, где каждый пакет находится в своем собственном модуле, полна маленьких модулей с небольшими поверхностями API. Эти крайности, конечно, смешны, но иллюстрируют проблему: все нужно уравновешивать.

ВАЖНАЯ ИНФОРМАЦИЯ

Итак, эти сигналы являются всего лишь сигналами — вы и ваша команда всегда должны руководствоваться собственным мнением на основе информации, которую они предоставляют. Но ваши модульные декларации могут помочь с этим. Если они становятся сложными и постоянно нуждаются во многих изменениях, то пытаются что-то сказать. Слушайте их.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

15.2.3. Позаботьтесь о декларациях модуля

При создании модульного проекта декларации модулей — это, пожалуй, самые важные файлы `.java` в кодовой базе. Каждый из них представляет собой полный JAR-файл, который, вероятно, будет состоять из десятков, сотен или даже тысяч исходных файлов. Более того, помимо простого представления модулей, модульные декларации определяют, как модуль взаимодействует с другими модулями.

Итак, вы должны позаботиться о декларациях модуля! Ниже представлены несколько моментов, на которые стоит обратить внимание:

- держите декларации в чистоте;
- комментируйте декларации;
- просматривайте декларации.

Рассмотрим их по порядку.

Чистые модульные декларации

Декларации модулей являются кодом и должны рассматриваться как таковые, поэтому убедитесь, что применяется общий стиль кода. Согласованный отступ, длина строки, положение скобок и т. д. — эти правила имеют такой же смысл для деклараций, как и для любого другого исходного файла.

Кроме того, я настоятельно рекомендую структурировать модульные декларации, а не помещать директивы в случайном порядке. Все декларации в JDK, а также в этой книге имеют следующий порядок.

1. `requires`, в том числе `static` и `transitive`.
2. `exports`.
3. `exports to`.
4. `opens`.
5. `opens to`.
6. `uses`.
7. `provides`.

JDK всегда помещает пустую строку между блоками, чтобы разделить их, — я делаю это, только когда директив несколько.

Пойдя дальше, можно определить, как упорядочить директивы в одном и том же блоке. Лексикографически это очевидный выбор, хотя для `requires` сначала лучше перечислить внутренние зависимости, а затем внешние.

ПРИМЕЧАНИЕ

Что бы вы ни решили, если у вас есть документ, определяющий стиль кода, то запишите это решение. Располагая IDE, инструментом для сборки или анализатором кода, проверьте их все, что будет даже лучше для вас. Попробуйте ускорить инструменты, чтобы они могли автоматически проверять или применять выбранный стиль.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Комментирование модульных деклараций

Мнения о документации кода, такие как Javadoc или встроенные комментарии, сильно разнятся, и это не тот случай, когда я могу аргументировать, что и почему важно. Но какова бы ни была позиция вашей команды в отношении комментариев, распространите ее на декларации модулей.

Если вам нравится идея, что у каждой абстракции есть предложение или небольшой абзац, объясняющий его значение и важность, то рассмотрите возможность добавления комментария Javadoc к каждому модулю:

```
/**
 * Aggregates service availability data points into statistics.
 */
module monitor.statistics {
    // ...
}
```

У JDK есть такой или даже более длинный комментарий на каждый модуль.

Даже если вам не нравится записывать действия модуля, большинство людей согласны с тем, что документирование того, почему было принято конкретное решение, имеет ценность. В декларации модуля это может означать добавление встроенного комментария:

- для необязательной зависимости, чтобы объяснить, почему модуль может отсутствовать;
- к квалифицированному экспорту с целью обосновать, почему он не является открытым API, но все еще был доступен для определенных модулей;
- к открытому пакету, объясняя, какие структуры планируют получить к нему доступ.

В JDK вы иногда найдете такие комментарии в *jdk.naming.rmi*:

```
// temporary export until NamingManager.getURLContext uses services
exports com.sun.jndi.url.rmi to java.naming;
```

В целом моя рекомендация такова: каждый раз в момент принятия решения, которое не сразу очевидно, добавьте комментарий. Всякий раз, когда рецензент спрашивает, почему были внесены некие изменения, добавляйте комментарий. Это может помочь вашим коллегам-разработчикам или вам самим спустя два месяца.

ВАЖНАЯ ИНФОРМАЦИЯ

Модульные декларации предоставляют новую возможность. Никогда прежде не было так легко правильно документировать отношения артефактов проекта в коде.

Просмотр деклараций модулей

Декларации модулей — центральное представление вашей модульной структуры, и их изучение должно быть неотъемлемой частью любого вида анализа кода, ко-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

торый вы делаете. Будь то просмотр изменений перед фиксацией или открытием запроса на слияние, компоновка кода после сеанса парного программирования или во время формальной проверки — каждый раз, проверяя тело кода, обратите особое внимание на `module-info.java`.

- Действительно ли необходимы добавленные зависимости? Соответствуют ли они основной архитектуре проекта? Должны ли они быть выставленными с `requires transitive`, поскольку их типы используются в API модуля?
- Если зависимость необязательна, то готов ли код обрабатывать ее отсутствие во время выполнения? Существуют ли эффекты, такие как отсутствие переходных зависимостей, для которых необязательная зависимость подразумевает читабельность?
- Может ли новая зависимость быть заменена сервисом?
- Действительно ли необходим добавленный экспорт? Все ли открытые классы во вновь экспортированных пакетах готовы для публичного использования, или их нужно перемешать, чтобы уменьшить поверхность API?
- Если экспорт квалифицирован, то имеет ли это смысл, или это просто отговорка, чтобы получить доступ к API, который никогда не должен был быть публичным?
- Используется ли тип как сервис, предназначенный для того, чтобы быть неотъемлемой частью инфраструктуры приложения?
- Были ли внесены какие-либо изменения, которые могут негативно повлиять на нижестоящих потребителей, не являющихся частью процесса сборки? (Подробнее см. в подразделе 15.2.4.)
- Является ли декларация модуля разработанной и прокомментированной в соответствии с требованиями команды?

Тщательный анализ особенно важен, поскольку IDE предлагают быстрые исправления, которые позволяют разработчикам редактировать декларации на расстоянии, экспортируя пакеты или добавляя зависимости с помощью простой команды. Я ценю эти функции, но они делают небрежное редактирование более вероятным; таким образом, тем более важно гарантировать, что ничто не подкрадется незамеченным.

ПРИМЕЧАНИЕ

Если у вас есть руководство по проверке кода, контрольный список или любой другой документ, который помогает поддерживать высокое качество кода, то можно добавить в него пункт о модульных декларациях.

Вложение времени в рассмотрение дескрипторов модуля может показаться большой работой. Для начала я бы поспорил, так ли уж она *велика*, особенно по сравнению с усилиями, направленными на разработку и анализ остальной части кода. Что более важно, я не расцениваю просмотр дескрипторов как дополнительную задачу — вместо этого я вижу новую возможность.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

ВАЖНАЯ ИНФОРМАЦИЯ

Никогда еще не было так легко анализировать и описывать структуру проекта. И это не рисунок на белой доске, который был сфотографирован и загружен в документацию команды несколько лет назад; нет, я говорю о реальной сделке, о реальных отношениях между артефактами. Декларации модуля показывают голую реальность, а не устаревшие благие намерения.

15.2.4. Взлом кода путем редактирования модульных деклараций

Как и в любом другом исходном файле, изменение декларации модуля может иметь непреднамеренные и, возможно, разрушительные последствия для другого кода. Более того, данная декларация является дистилляцией открытого API модуля и поэтому оказывает гораздо более сильное влияние, чем любой случайный класс.

Если вы разрабатываете приложение и все потребители модуля — часть одного и того же процесса сборки, то критические изменения не могут остаться незамеченными. Даже для фреймворков и библиотек такие изменения можно обнаружить только с помощью тщательных интеграционных тестов.

ВАЖНАЯ ИНФОРМАЦИЯ

Тем не менее это помогает понять, какие изменения чаще вызывают проблемы, а какие обычно являются доброкачественными. Ниже представлен список наиболее проблемных из них.

1. Новое имя модуля.
2. Уменьшение числа экспортируемых пакетов.
3. Различные предоставляемые сервисы.
4. Редактирование зависимостей.

Ниже будет показано, что все эти изменения могут вызвать ошибки компиляции или непредвиденное поведение во время выполнения в последующих проектах. В связи с этим они всегда должны рассматриваться как критические изменения, поэтому в случае использования семантического управления версиями важен основной вариант версии. То есть внесение других изменений в декларации модуля также не может вызвать проблем, но они гораздо менее вероятны; итак, сосредоточимся на этих четырех.

Влияние нового имени модуля

Изменение имени модуля немедленно сломает все модули, которые зависят от него, — их нужно будет обновить и перестроить. Это наименьшая из возможных проблем.

Гораздо более опасен модульный алмаз смерти (см. подраздел 3.2.2), который может появиться, когда какой-то проект транзитивно зависит от модуля дважды:

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

со старым именем и с новым. Этот проект будет испытывать трудности с включением новой версии в свою сборку, и, возможно, придется прибегнуть к отказу от обновления только из-за измененного имени.

Помните о данном обстоятельстве и постарайтесь свести к минимуму переименования. Возможно, все еще придется делать это время от времени, и в таком случае можно попытаться смягчить последствия, создав модуль агрегатора со старым именем (объяснено в подразделе 11.1.5).

Влияние экспорта меньшего количества пакетов

Должно быть очевидно, почему «неэкспортируемые» пакеты вызывают проблемы: любой модуль, использующий типы в этих пакетах, не сможет получить к ним доступ во время компиляции и выполнения. Если вы хотите пойти по данному пути, то необходимо сначала отказаться от этих пакетов и типов, чтобы дать пользователям время отойти от них, прежде чем они будут удалены.

Это в полной мере относится лишь к простым директивам `exports`:

- квалифицированные экспорты обычно экспортируются только в другие модули, которыми вы управляете, вероятно являющиеся частью сборки и, таким образом, обновляющиеся одновременно;
- открытые пакеты обычно ориентированы на конкретный фреймворк или фрагмент кода, который предназначен для их рефлексии. Данный код редко входит в состав модулей пользователей, поэтому на него не повлияет закрытие пакета.

Вообще говоря, я бы не думал, что удаление квалифицированных экспортов или открытых пакетов — серьезное изменение. Тем не менее конкретные сценарии могут идти вразрез с этим эмпирическим правилом, вследствие чего следите за ними и продумывайте путь при внесении таких изменений.

Влияние добавления и удаления сервисов

Ситуация с сервисами немного менее ясна. Как описано в подразделе 10.3.1, потребители сервисов всегда должны быть готовы к отсутствию поставщиков; точно так же им не следует ломаться, когда неожиданно возвращается дополнительный поставщик. Но из вышеперечисленного действительно лишь то, что приложения не должны падать, если загрузчик сервисов вернул неверное количество поставщиков.

Вполне возможно, даже вероятно, что приложение ведет себя неправильно, поскольку сервис был в одной версии, а в другой его нет. А так как привязка сервиса происходит во всех модулях, это может даже повлиять на код, который напрямую не зависит от данного модуля.

Влияние редактирования зависимостей

Последний пункт в списке, зависимости во всех их формах, также является серой областью. Начнем с `requires transitive`. В подразделе 11.1.4 объясняется, что пользователи должны полагаться только на зависимость, которую вы позволяете

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

им читать, если они задействуют ее в непосредственной близости от вашего модуля. Учитывая предположение, что вы перестали выставлять типы зависимостей, а пользователи обновили свой код, удаление `transitive` из директивы `exports` не должно влиять на них.

С другой стороны, они могут не знать об этой рекомендации или не учитывать ее. Как следствие, чтобы они не могли прочитать данную зависимость, им все равно придется обновлять и перестраивать свой код. Следовательно, я все еще считаю это переломным изменением.

Вдобавок можно придумать сценарии, в которых удаление или даже добавление других зависимостей чревато проблемами, даже если это не должно наблюдаться извне данного модуля:

- добавление или удаление простой директивы `requires` изменяет разрешение необязательных зависимостей и привязку сервиса;
- если сделать зависимость необязательной (или пойти другим путем), то можно изменить и то, какие модули будут включены в граф.

Поэтому, несмотря на то что `requires` и `requires static` могут изменить граф модулей и таким образом повлиять на модули, которые абсолютно не связаны с нужным модулем, это нечастый случай. По умолчанию я не считаю такие изменения разрушительными.

ПРИМЕЧАНИЕ

Хотя все представленное выше может показаться ужасным и сложным, это не более чем случай, когда вы редактируете классы, являющиеся частью общедоступного API. Вашей интуиции в том, как изменения в декларации модуля влияют на другие проекты, пока недостаточно. Она придет со временем.

15.3. Технологический ландшафт

Впервые представив систему модулей в разделе 1.4, я подумал, что у вас может возникнуть несколько вопросов о механизмах ее связи с остальной частью экосистемы. Как вы помните, эти вопросы появлялись так.

- Разве Maven, Gradle и другие еще не управляют зависимостями?
- Как насчет OSGi? Почему бы просто не использовать его?
- Не является ли система модулей избыточной во времена, когда все пишут микросервисы?

Я отвечу на них совсем скоро, но сначала хочу познакомить вас с тем, что Java 9+ может предложить помимо системы модулей. В конце концов, эти преимущества входят в один пакет, и если вы скептически относитесь к одному из них, то, возможно, преимущества другого могут повлиять на вас.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

15.3.1. Maven, Gradle и другие инструменты сборки

Экосистеме Java повезло иметь несколько эффективных, проверенных в бою инструментов сборки, таких как Maven и Gradle. Конечно, они неидеальны, но строят Java-проекты уже более десяти лет, поэтому в них, безусловно, есть что-то стоящее.

Как следует из названия, основная задача инструмента сборки — создание проекта, включающего в себя компиляцию, тестирование, сборку и распространение. Хотя система модулей затрагивает многие из этих шагов и требует неких изменений в инструментах, она не добавляет платформе никаких возможностей, которые заставили бы ее конкурировать с ними в данной области. Поэтому, когда дело доходит до *создания* проекта, связь между платформой Java и ее инструментами сборки остается практически неизменной.

Инструменты сборки на Java 9+

Я не могу говорить обо всех инструментах сборки, но Maven и Gradle уже обновлены для правильной работы с Java 9+ и системой модулей. Изменения в основном внутренние, и для создания модульных JAR-файлов вместо простых JAR не требуется ничего, кроме добавления `module-info.java` в исходную папку. Они берут файл оттуда и в основном просто делают правильные вещи.

Подробнее о том, как выбранный вами инструмент сборки взаимодействует с системой модулей или другими новыми функциями Java (например, JAR с несколькими выпусками, см. приложение Д), смотрите в его документации. Я хочу упомянуть одно: вероятно, придется добавить некоторые параметры командной строки при переходе на Java 9+, поэтому может потребоваться время, чтобы разобраться в том, как это сделать.

Если хотите узнать больше о Gradle, то ознакомьтесь с практическим руководством *Gradle in Action* (Бенджамин Мушко, 2014, издательство Manning, www.manning.com/books/gradle-in-action). К сожалению, я не могу порекомендовать ни одну книгу о Maven, которую мне удалось хотя бы пролистать.

Управление зависимостями

Системы сборки обычно выполняют другую задачу, и теперь и Java 9+ выполняет ее: управление зависимостями. Как указано в разделе 3.2, надежная конфигурация направлена на то, чтобы зависимости присутствовали и были однозначными, а приложение стало более стабильным, — Maven или Gradle сделают то же самое. Означает ли это, что система модулей заменяет инструменты сборки? Или же игра устарела и данные функции бесполезны? На первый взгляд кажется, будто система модулей дублирует функциональность инструментов сборки; но, присмотревшись, можно увидеть, что пересечение совсем невелико.

Для начала система модулей не может однозначно идентифицировать или определить местонахождение артефактов. Наиболее примечательно то, что у нее нет понятия версий, то есть нужно иметь в виду, что при наличии нескольких разных

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

версий одного и того же артефакта она не сможет выбрать правильную. Такая ситуация приведет к ошибке именно потому, что неоднозначна.

Многие проекты выбирают имя модуля, которое может быть уникальным (например, изменение доменного имени, с которым связан проект). Однако не существует такого экземпляра, как Maven Central, который бы гарантировал это, что делает имя модуля недостаточным для уникальной идентификации зависимости. Если говорить об удаленных репозиториях, таких как Maven Central, система модулей не имеет возможности подключаться к ним. Таким образом, невзирая на то что и она, и инструменты сборки управляют зависимостями, система модулей работает на уровне слишком абстрактном, чтобы заменить инструменты сборки.

Однако у систем сборки есть существенный недостаток: они гарантируют наличие зависимостей во время компиляции и даже могут доставить их прямо в нужное место, но не управляют запуском приложения. Если инструмент не знает о косвенно требуемой зависимости (из-за использования `provided` от Maven или `compileOnly` от Gradle) или библиотека теряется на пути от сборки к запуску, то вы узнаете об этом лишь во время выполнения, скорее всего, из-за сбоя. С другой стороны, система модулей управляет прямыми и переходными зависимостями во время не только компиляции, но и выполнения, обеспечивая надежную конфигурацию на всех этапах. Кроме того, она лучше приспособлена для обнаружения неоднозначностей, например повторяющихся артефактов или артефактов, содержащих одинаковые типы. Таким образом, даже при увеличении масштаба управления зависимостями обе технологии различны; единственное совпадение состоит в том, что обе перечисляют зависимости в той или иной форме.

Инкапсуляция, сервисы и ссылки

Отойдя от управления зависимостями, мы быстро находим функции системы модулей, с которыми инструменты сборки не могут конкурировать. Наиболее примечательна строгая инкапсуляция (см. раздел 3.3), позволяющая библиотекам скрывать детали реализации от другого кода во время компиляции и во время выполнения, — то, о чем Maven или Gradle не могут и мечтать. Потребуется некоторое время, чтобы привыкнуть к этой строгости. Однако в долгосрочной перспективе JDK, фреймворки, библиотеки и даже большие приложения выиграют от четкого различия поддерживаемых и внутренних API и защиты последних от случайного использования. На мой взгляд, даже одна строгая инкапсуляция стоит того, чтобы перейти к системе модулей.

При рассмотрении более расширенных функций две из них представляются особенно интересными как недоступные для инструментов сборки. Во-первых, система модулей может работать как реестр сервисов в *шаблоне локатора сервисов*, позволяя отделять артефакты и реализовывать приложения, облегчающие использование плагинов (см. главу 10). Во-вторых, это возможность связать нужные модули в автономный образ среды выполнения, что позволяет сделать развертывание более гибким и простым (см. главу 14).

Таким образом, за исключением небольшого дублирования в управлении зависимостями, инструменты сборки и система модулей не конкурируют, а должны

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

рассматриваться как дополнения друг для друга. На рис. 15.4 показана эта взаимосвязь.

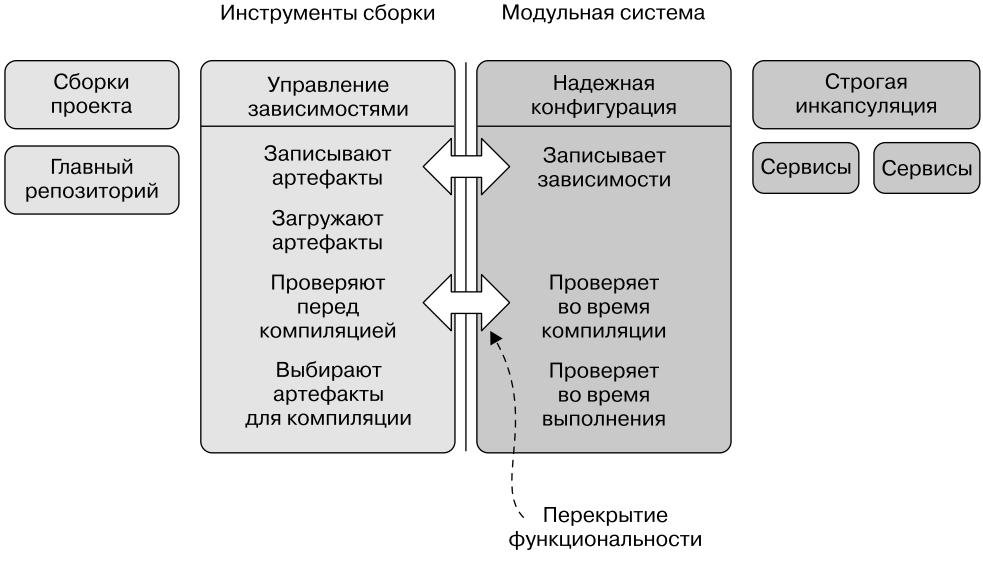


Рис. 15.4. Инструменты сборки *(слева)* и система модулей *(справа)* имеют очень разные наборы функций. Единственное сходство заключается в том, что обе записывают зависимости (инструменты сборки по глобально уникальным идентификаторам плюс версии; JPMS только по именам модулей) и могут проверять их для компиляции. Их обработка зависимостей весьма разнится, и за исключением этого у них практически нет ничего общего

15.3.2. OSGi

Инициатива Open Service Gateway (OSGi) — обозначение как для организации (OSGi Alliance), так и для создаваемой ею спецификации. Вдобавок оно несколько неточно применяется к различным реализациям данной спецификации, и именно так я использую ее в этом разделе.

OSGi — это система модулей и сервисная платформа, основанная на виртуальной машине Java, которая разделяет часть своего набора функций с JPMS. Если вы знаете кое-что о OSGi или уже пользовались ей, то можете спросить, как она соотносится с новой системой модулей Java и, возможно, заменяется ли ею. Но вы также можете удивиться, почему вообще последняя была разработана — разве Java не мог просто применить OSGi?

ПРИМЕЧАНИЕ

Данный подраздел может быть немного сложным, если вы знаете OSGi только понаслышке, — это не проблема, вследствие чего он необязателен для чтения. Если вы все еще хотите продолжить, то начните с представления, что OSGi похожа на систему модулей. Ниже будут освещены некоторые важные различия.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Я не эксперт по OSGi, но во время своего исследования пролистал *OSGi in Depth*, и мне понравилось (Александр де Кастро Альвз, 2011, издательство Manning, www.manning.com/books/osgi-in-depth). Подумайте об этом, если вам нужно больше, чем может предложить система модулей платформы Java.

Почему JDK не использует OSGi?

Почему JDK не использует OSGi? Технический ответ на этот вопрос сводится к тому, как OSGi реализует свой набор функций. Он в значительной степени опирается на загрузчики классов, которые мы кратко обсудили в разделе 1.2 и подразделе 1.3.4 и для которых OSGi создает собственные реализации. Она использует один загрузчик классов на связку (в OSGi модули называются *связками*) и таким образом контролирует, например, какие классы можно увидеть в связке (для реализации инкапсуляции) или что происходит, когда выгружается связка (OSGi позволяет это, подробнее — позже).

То, что может показаться технической деталью, имеет далекоидущие последствия. До JPMS Java не накладывал никаких ограничений на использование загрузчиков классов и применение API рефлексии для доступа к классам по имени было обычной практикой.

Если бы JPMS требовала особой архитектуры загрузчика классов, то Java 9+ радикально сломал бы JDK, многие существующие библиотеки и фреймворки и критический код приложения. Java 9+ по-прежнему создает проблемы миграции, но несовместимое изменение API загрузчика классов будет еще более разрушительным и не заменит указанные проблемы, а превзойдет их. Вследствие этого JPMS работает ниже загрузчиков классов, как показано на рис. 15.5.

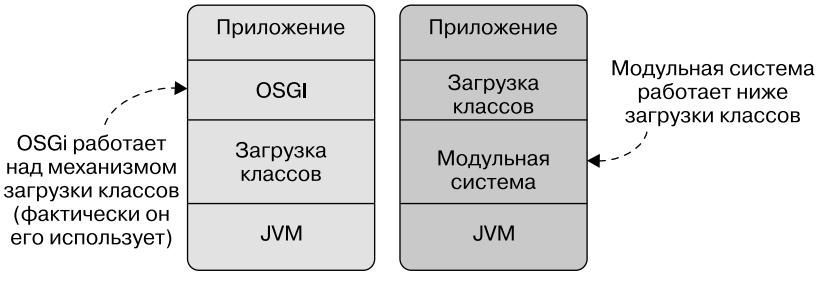


Рис. 15.5. OSGi *(слева)* построена поверх JVM, что вынудило ее использовать существующие функциональные возможности, в основном инфраструктуру загрузки классов, для реализации своего набора функций. С другой стороны, система модулей *(справа)* была реализована в JVM и работает при загрузке ниже класса, благодаря чему системы, построенные на ней, работают как прежде

Другое следствие использования загрузчиков классов для изоляции модулей состоит в том, что, хотя OSGi применяет их с целью уменьшения *видимости* классов, они не могут уменьшить *доступность*. Что я имею в виду под этим? Скажем, связка *lib* содержит тип `Feature` из пакета, который не экспортируется. Затем OSGi убеждается в отсутствии у кода в другой связке *app* возможности видеть `Feature`; это

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

означает, например, что `Class.forName("org.lib.Feature")` сгенерирует исключение `ClassNotFoundException` (`Feature` не виден).

Но теперь предположим, что в *lib* есть API, который возвращает `Feature` как объект, и в этом случае *app* может получить экземпляр класса. Затем *app* может вызвать `featureObject.getClass().newInstance()` и создать новый экземпляр компонента (`Feature` доступен).

Как обсуждалось в разделе 3.3, JPMS хочет обеспечить надежную инкапсуляцию, а того, что может предложить OSGi, недостаточно. Если вы создадите ситуацию, подобную предыдущей, с двумя JPMS-модулями, *app* и *lib* и типом `Feature`, который *lib* содержит, но не экспортирует, то *app* может успешно получить экземпляр класса с помощью `Class.forName("org.lib.Feature")` (видимый), но не может вызвать `newInstance()` для него (недоступный). В табл. 15.1 сопоставляются различия OSGi и JPMS.

Таблица 15.1. Видимость OSGI и ограничения доступности JPMS

Действие	**OSGi**	**JPMS**
Ограничивает видимость (сбой Class::forName)	✔	✘
Ограничивает доступность (сбой Class::newInstance)	✘	✔

Может ли JPMS заменить OSGi?

Может ли JPMS заменить OSGi? Нет.

JPMS была первоначально разработана для модульного JDK. Она охватывает все основы модульности — некоторые из них, такие как инкапсуляция, возможно, даже лучше, чем OSGi, — но OSGi обладает множеством функций, в которых JPMS не нуждается и, следовательно, не имеет.

Иными словами, благодаря OSGi, в частности ее стратегии загрузчика классов, можно иметь один и тот же полностью определенный тип в нескольких пакетах. Она также позволяет запускать разные версии одного и того же пакета одновременно. В этой связи с помощью OSGi экспорт и импорт могут быть версионными, что позволяет пакетам указывать, какая у них версия и какие версии должны быть у их зависимостей. Если один и тот же пакет требуется в двух разных версиях, то OSGi может заставить это работать.

Другое интересное отличие состоит в том, что в OSGi связка обычно выражает зависимости от пакетов, а не от связок. Возможны оба варианта, однако первый используется по умолчанию. Это делает зависимости более надежными в отношении замены или рефакторинга связок, поскольку не имеет значения, откуда поступает пакет. (В JPMS, с другой стороны, пакет должен находиться в одном из обязательных модулей, поэтому перемещение пакета в другой модуль или замена одного модуля другим с тем же API может вызвать проблемы.)

Большой набор функций OSGi вращается вокруг динамического поведения, где ясно показываются его корни как сервисы Интернета вещей и реализация через загрузчики классов обеспечивает эффективные возможности. OSGi позволяет пакетам появляться, исчезать и даже обновляться во время выполнения, предоставляя API, благодаря которому зависимости могут реагировать соответствующим образом.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Это отлично подходит для приложений, работающих на нескольких устройствах, но может пригодиться и для систем с одним сервером, желающим сократить время простоя до минимума.

Суть в следующем: если ваш проект уже использует OSGi, то высоки шансы, что вы полагаетесь на функции, которых нет у JPMS. В этом случае нет причин переключаться на встроенную систему модулей Java.

OSGi устраняет JPMS?

OSGi устраняет JPMS? Нет.

Хотя то, что я говорил секунду назад, звучит очень похоже на то, что OSGi лучше, чем JPMS для каждого варианта использования, OSGi никогда не встречала широкого применения. Она заняла свою нишу и добилась успеха, но никогда не была технологией по умолчанию (в отличие от IDE, инструментов сборки и ведения журналов, например).

Основная причина этого отсутствия широкого применения — сложность. Является ли она воспринимаемой или реальной, присуща ли она модульности или случайна для OSGi — вторично по отношению к тому факту, что большинство разработчиков рассматривают сложность OSGi как причину, по которой она не используется по умолчанию.

JPMS находится в другом положении. Для начала ее сокращенный набор функций (в частности, отсутствие поддержки версий и зависимости от модулей, а не пакетов) делает ее менее сложной. Кроме того, она выигрывает от встраивания в JDK. Все Java-разработчики в некоторой степени знакомы с JPMS, и более опытные специалисты, в частности, изучат, как это может помочь им в их проектах. Такое более интенсивное использование будет способствовать и хорошей интеграции инструментов.

Итак, если команда уже обладает навыками и инструментами и работает над JPMS, то почему бы не пройти весь путь и не модуляризовать приложение целиком? Этот шаг основывается на существующих знаниях, требует меньше дополнительных сложностей и не нуждается в новых инструментах, но дает много преимуществ.

В конце концов, даже OSGi выиграет от JPMS, поскольку Java 9+ положит модульность на карту так же, как Java 8 сделал это с функциональным программированием. Оба релиза открывают для новых разработчиков Java новые идеи и учат их совершенно новому набору навыков. В какой-то момент, когда проект выигрывает от функционального программирования или более эффективной модульности, его разработчики преодолевают достаточную кривую обучения, чтобы оценить и, возможно, использовать «реальную вещь».

Совместимы ли JPMS и OSGi?

Совместимы ли JPMS и OSGi? В некотором смысле. Приложения, созданные с помощью OSGi, могут работать поверх Java 9+, как и в более ранних версиях. (Точнее, будут работать в *безымянном модуле*, который подробно описан в разделе 8.2.) OSGi не предпринимает никаких усилий по миграции, но код приложения сталкивается с теми же проблемами, что и другие кодовые базы.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

С другой стороны, приговор еще не вынесен. Вопрос, позволит ли OSGi связывать пакеты с модулями JPMS, пока остается открытым. На данный момент OSGi не использует возможности JPMS и продолжает реализовывать свои функции самостоятельно. Неясно также, стоит ли адаптация OSGi к JPMS значительных затрат на разработку.

15.3.3. Микросервисы

Отношения между системой модулей и архитектурой микросервисов имеют два совершенно разных аспекта.

- Являются ли микросервисы и система модулей конкурентными? Как они сравниваются?
- Беспокоит ли вас система модулей, если вы пользуетесь микросервисами?

Мы рассмотрим оба аспекта в этом подразделе.

Если вы не знакомы с архитектурой микросервисов, то можете смело пропустить данный фрагмент. Для желающих узнать больше могу сказать, что есть множество отличных книг по микросервисам. В подтверждение своих заявлений я просмотрел *Microservices in Action* (Морган Брюс и Пауло А. Перейра, 2018, издательство Manning, www.manning.com/books/microservices-in-action) и могу рекомендовать его.

Микросервисы против JPMS

В целом справедливо сказать, что преимущества системы модулей имеют тем больший эффект, чем крупнее проект. Итак, когда все говорят о микросервисах, не является ли система модулей для больших приложений всего лишь красивой оберткой? Ответ зависит от того, сколько проектов в конечном итоге будет структурировано как микросервисы, и это, конечно, само по себе является темой для долгих дискуссий.

Одни считают, что за микросервисами будущее и рано или поздно все проекты начнутся именно так — все это микросервисы! Если вы находитесь в данном лагере, то все равно можете реализовать свои сервисы на Java 9+, и система модулей повлияет и на ваш проект тоже, но, конечно, в гораздо меньшей степени, чем на монолитные проекты. Мы обсудим это ниже.

Другие разработчики более осторожны в высказываниях. Как и все архитектурные стили, микросервисы имеют и преимущества, и недостатки, и между ними необходимо найти компромисс с учетом требований проекта. Микросервисы особенно ярко проявляют себя в довольно сложных проектах, которые должны выдерживать высокие нагрузки, где их способность к масштабированию практически не имеет себе равных.

Однако за такую масштабируемость нужно платить сложностью операций, поскольку запуск множества сервисов требует гораздо больше знаний и инфраструктуры, чем размещение нескольких экземпляров одного и того же сервиса на балансировщике нагрузки. Еще один недостаток состоит в том, что исправление

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

неправильного определения границ сервисов, которое тем более вероятно, чем меньше команда знает о домене, обходится дороже в микросервисах, чем в монолите.

Критическое наблюдение заключается в том, что цена за сложность (Мартин Фаулер называет это *премией за микросервисы*) всегда должна быть оплачена, но выгоды можно получить, только когда проект достаточно велик. Данный фактор убедил многих разработчиков и архитекторов в том, что большинство проектов должны начинаться как монолит и переходить к разделению сервисов, возможно в конечном итоге заканчивая микросервисами, в зависимости от обстоятельств.

Мартин Фаулер, например, приводит следующие мнения своих коллег (в martinfowler.com/bliki/MonolithFirst.html):

> Не следует начинать новый проект с микросервисов, даже если вы уверены, что ваше приложение будет достаточно большим, чтобы оно того стоило. [...] Логичный путь — тщательно спроектировать монолит, обращая внимание на модульность в программном обеспечении как на границах API, так и в способе хранения данных. Сделайте это хорошо, и перейти к микросервисам будет относительно просто.

К настоящему времени эти фразы должны быть знакомы: тщательный дизайн, модульность, границы — все это свойства, которые продвигает система модулей (см. раздел 1.5). В микросервисной архитектуре зависимости сервисов должны быть четкими (например, надежная конфигурация) и идеально отделенными (API загрузчика сервисов); более того, все запросы должны проходить через общедоступные API (строгая инкапсуляция). Осторожное использование системы модулей может заложить основу для успешного перехода на микросервисы, если и когда придет время. На рис. 15.6 показана важность этой тщательной разработки.

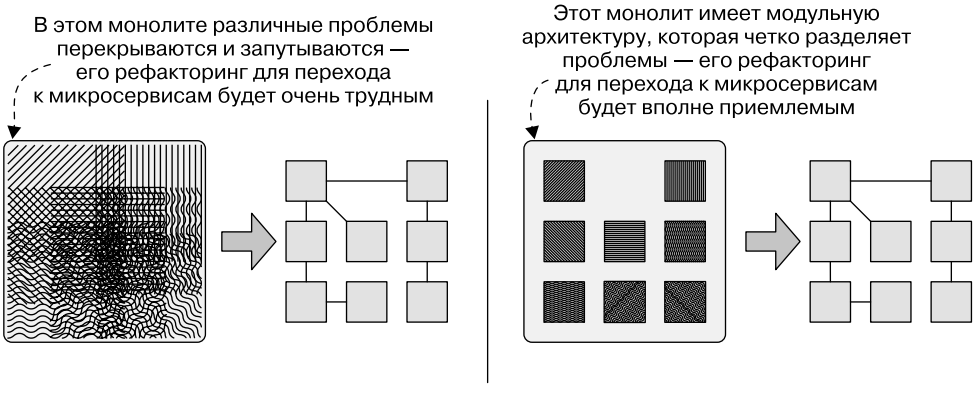


Рис. 15.6. С учетом двух гипотетических миграций монолитного приложения на микросервисы вы бы предпочли начать со значительного квадрата грязи *(слева)* или с правильно модуляризованной кодовой базы *(справа)*?

Несмотря на то что система модулей ориентирована на более крупные проекты, даже небольшие сервисы могут выиграть от использования модулей.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Микросервисы с JPMS

Если проект состоял из микросервисов и вы внедряете часть из них на основе Java 9+, поскольку хотите получить выгоду от повышения безопасности и производительности, то обязательно будете взаимодействовать с системой модулей: она работает в JVM, которая выполняет ваш код. Одно из последствий этого таково: потенциально критические изменения, обсуждаемые в главах 6 и 7, по-прежнему применяются к рассматриваемым сервисам и требуют исправления. Кроме того, вероятно, что большинство зависимостей со временем превратятся в модули, но согласно описанному в подразделе 8.1.3 это не заставляет вас упаковывать артефакты как модули.

Если вы решите сохранить все JAR-файлы в пути к классам, то строгая инкапсуляция между ними не будет применяться. Таким образом, в этом наборе JAR-файлов доступ к внутренним API, как и рефлексия, например, из фреймворков в ваш код, будут продолжать работать. В данном сценарии воздействие разработчиков на систему модулей ограничено изменениями, внесенными в JDK.

Другой путь, которым можно было бы воспользоваться, — это применение сервисов и зависимостей в качестве модулей, после чего произойдет полная интеграция в систему модулей. Из его различных преимуществ наиболее подходящим может оказаться масштабируемость платформы (кратко описанная в подразделе 1.5.5 и подробно изученная в главе 14), которая позволяет задействовать `jlink`.

С помощью `jlink` можно создать небольшой образ среды выполнения с правильным набором платформенных модулей для поддержки приложения, *включая модули*, которые могут сократить размер образа до 80 %. Кроме того, при связывании необходимых модулей `jlink` может анализировать байт-код, зная, что он видит все приложение и, следовательно, может применять более агрессивные оптимизации, приводящие к еще меньшим размерам образа и немного улучшенной производительности. Вы получаете и другие преимущества: например, уверенность в том, что используете только общедоступные API своих зависимостей.

15.4. Размышления о модульной экосистеме

Java 9+ — массовый релиз. Несмотря на отсутствие новых языковых функций, он содержит множество эффективных улучшений и дополнений. Но все они затмеваются системой модулей платформы Java. Это легко ожидаемая и наиболее спорная функция Java 9+, не в последнюю очередь из-за проблем миграции, которые она вызывает.

Несмотря на порой нелегкое начало на пути к модульному будущему, известные библиотеки и платформы быстро стали поддерживать Java 9+, и с тех пор не было никаких признаков замедления этой тенденции. Как насчет старых, менее поддерживаемых проектов? Хоть некоторые и могут найти новых сопровождающих, даже если просто заставить их работать над текущей версией Java, длинный хвост проектов Java может уменьшиться.

Это наверняка вызовет недовольство некоторых разработчиков, чьи кодовые базы зависят от таких проектов. Оно и понятно — никто не любит менять рабочий код

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

без видимой выгоды. В то же время уход части сотрудников даст другим проектам возможность увеличить количество пользователей. А кто знает? Может быть, они в конечном итоге увидят выгоду от переключения.

Когда большая волна обновления до Java 9+ будет позади и проекты начнут повышать свои базовые показатели до Java 9+, вы увидите рост общедоступности модульных JAR-файлов. Благодаря тому, что модульная система поддерживает поэтапную и децентрализованную модуляризацию, данный процесс требует сравнительно небольшой координации между проектами. Это также дает возможность начать модуляризацию проекта прямо сейчас.

Для чего? В отличие от более ярких функций, таких как лямбда-выражения и потоки в Java 8 или вывод типа локальной переменной в Java 10, влияние системы модулей на кодовую базу будет незначительным. Вы не сможете просмотреть несколько строк кода и довольствоваться его красотой. Вы не заметите внезапно, что писать код стало веселее.

Нет, преимущества системы модулей находятся на другом конце спектра. Вы будете обнаруживать больше ошибок на ранней стадии благодаря надежной конфигурации. Вы избежите их благодаря лучшему пониманию архитектуры проекта. Вы не сможете легко запутать код и не будете случайно зависеть от внутренних зависимостей.

Это капризные части разработки программного обеспечения, которые улучшит JPMS. Система модулей не панацея: все равно придется потрудиться, чтобы правильно спроектировать и расположить артефакты. Но с системой модулей под рукой и данное усилие столкнется с меньшим количеством ловушек на пути и получит больше наград.

Поскольку все больше и больше артефактов экосистемы становятся модульными, данный эффект будет только усиливаться, пока в один прекрасный день мы не спросим себя, как вообще раньше писали код без системы модулей. Каково это было во времена, когда JVM превратила наш тщательно разработанный граф зависимостей в комок грязи?

Будет странно думать о чем-то другом. Так же с написанием класса Java без `private`. Можете представить, на что это было бы похоже?

Резюме

- Тщательно спроектируйте свою систему модулей.
- Микросервисы и JPMS дополняют друг друга.
- OSGi и JPMS тоже дополняют друг друга.

А теперь — большое спасибо за то, что прочитали эту книгу. Было приятно писать для вас. Я уверен, мы увидимся снова!

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Приложения

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

A Резюмируем путь класса

Книга, в которой обсуждается система модулей, конечно же, фокусируется на пути модуля (см. раздел 3.4). Но путь класса все еще полностью работоспособен, и, поскольку можно использовать его рядом с путем модуля, он играет важную роль во время инкрементной модуляризации. Другими словами, все еще стоит знать, как он функционирует.

A.1. Использование пути к классам для загрузки JAR-файлов приложений

Определение: путь класса

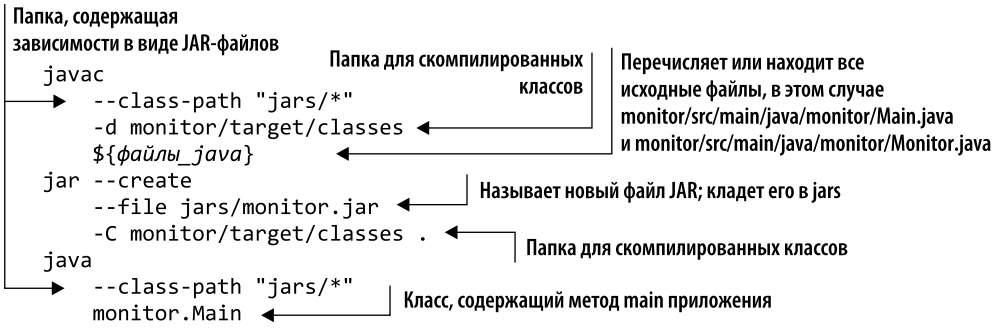
Путь класса — это концепция, связанная с компилятором и виртуальной машиной. Они используют его для той же цели: поиска в перечисленных JAR-файлах типов, которые им требуются, но не имеются в JDK. (Он также может применяться с файлами классов, но вы можете игнорировать этот случай при изучении системы модулей.)

Рассмотрим демонстрационное приложение *ServiceMonitor* из этой книги в качестве примера. Оно состоит из нескольких подпроектов и имеет ряд зависимостей. В этом сценарии все подпроекты, кроме последнего, *monitor*, уже созданы и находятся в каталоге `jars`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

В листинге А.1 показано, как скомпилировать, упаковать и запустить приложение, используя путь к классам. За исключением новых вариантов некоторых параметров командной строки (например, с помощью `--class-path`, а не `--classpath`), это те же команды, что и до Java 9.

Листинг А.1. Компиляция, упаковка и запуск с помощью пути к классам



И компилятор, и среда выполнения ищут в пути к классам нужные им типы. В целом они все же различаются.

- *Компилятору* требуются типы, на которые ссылается компилируемый код. Это прямые зависимости проекта или, точнее, те типы в прямых зависимостях, на которые ссылается компилируемый файл.
- *Виртуальная машина* запрашивает все типы, на которые ссылается исполняемый байт-код. В целом это прямые и косвенные зависимости проекта, но из-за ленивого подхода Java к загрузке классов их может быть значительно меньше. Требуются только типы, на которые ссылается код, выполняемый на самом деле, что означает потенциальное отсутствие зависимости, если код, использующий ее, не выполняется. JVM также позволяет коду искать JAR-ресурсы.

И `javac`, и `java` имеют параметры командной строки `-classpath`, `-cp` и, начиная с Java 9, `--class-path`. Как правило, они ожидают список файлов, но можно использовать пути и подстановочные знаки, которые затем распространяются на такой список.

А.2. Путь к классам, начиная с Java 9

ВАЖНАЯ ИНФОРМАЦИЯ

Что касается Java 9 (и более поздних версий), важно подчеркнуть, что путь к классам не уходит! Он работает точно так же, как и в более ранних версиях Java, и если приложения, скомпилированные в подобной версии, не сделали ничего проблемного (см. главы 6 и 7), то продолжат компиляцию в Java 9 и более поздних версиях с помощью тех же команд.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Принимая во внимание эту обратную совместимость, остается вопрос, как система модулей работает с типами в пути к классам. Короче говоря, все они оказываются в *безымянном модуле*, который система модулей преобразует на лету. Данный модуль обычен, но имеет ряд особенностей, одна из которых заключается в том, что он автоматически читает все разрешенные модули. Это также верно для модулей, попадающих в путь к классам, — они станут обрабатываться как обычные JAR-файлы, и их типы тоже будут в конечном итоге в безымянном модуле, игнорируя то, о чем говорит их модульная декларация. Безымянный модуль и модули в пути к классам — часть истории миграции, о которой подробно рассказывается в разделе 8.2.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Б

Обзор API рефлексии

Рефлексия позволяет коду проверять типы, методы, поля, аннотации и т. д. во время выполнения и откладывать решение о том, как их использовать, от времени компиляции до времени выполнения. С этой целью API рефлексии Java предлагает такие типы, как `Class`, `Field`, `Constructor`, `Method`, `Annotation` и др. С их помощью можно взаимодействовать с типами, которые не были известны во время компиляции: например, создавать экземпляры неизвестного класса и вызывать методы для них.

Рефлексия и варианты ее использования могут быстро стать сложными, и я не буду объяснять их подробно. Данное приложение предназначено дать общее представление о том, что такое рефлексия, как она выглядит в Java и для чего вы или ваши зависимости можете ее применять.

После этого вы будете готовы начать ею пользоваться или работать с более основательными учебниками, такими как курс *Reflection API* от Oracle по адресу docs.oracle.com/javase/tutorial/reflect. Что более важно, вы будете готовы понять изменения, вносимые системой модулей в отношении рефлексии, которые рассматриваются в подразделе 7.1.4 и особенно в главе 12.

Вместо того чтобы строить все с нуля, начнем с простого примера. Следующий фрагмент создает URL, преобразует его в строку и затем печатает его. Прежде чем перейти к рефлексии, я использовал простой Java-код:

```
URL url = new URL("http://codefx.org");
String urlString = url.toExternalForm();
System.out.println(urlString);
```

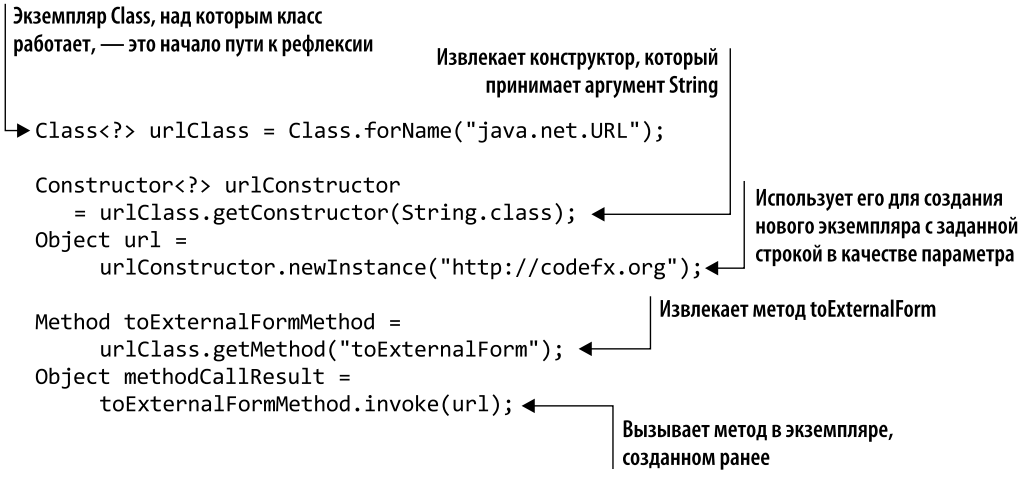
Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Во время компиляции (то есть во время написания кода) я решил, что хочу создать объект `URL` и вызвать в нем метод. Хоть это и не самый естественный способ решить данную задачу, можно разбить первые две строки на пять шагов.

1. Обратиться к классу URL.
2. Найти конструктор с одним строковым параметром.
3. Перейти по адресу `http://codefx.org`.
4. Найти метод `toExternalForm`.
5. Вызвать его на экземпляре `url`.

В листинге Б.1 показано, как реализовать эти пять шагов с помощью API рефлексии Java.

Листинг Б.1. Рефлексивное создание URL и вызов toExternalForm в нем



Использовать API рефлексии, конечно, более обременительно, чем непосредственно писать код. Но таким образом детали, которые раньше вставлялись в код (например, с помощью `URL` или любого выбранного метода), становятся строковым параметром. Как следствие, вместо того, чтобы останавливаться на `URL` и `toExternalForm` во время компиляции, вы можете решить, какой тип и метод выбрать позже, когда программа уже запущена.

Большинство случаев использования для этого происходит в «фреймворковых» средах. Подумайте, например, о JUnit, который хочет выполнить все методы, аннотированные `@Test`. После их обнаружения он задействует `getMethod` и `invoke` для вызова. Spring и другие веб-фреймворки действуют аналогично при поиске контроллеров и соответствия запросов. Расширяемые приложения, которые хотят загружать предоставленные пользователем плагины во время выполнения, — еще один пример применения.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Б.1. Основные типы и методы

Точка входа в API рефлексии — `Class::forName`. В своей простой форме этот статический метод принимает полностью определенное имя класса и возвращает для него экземпляр класса. Тот, в свою очередь, используется для получения полей, методов, конструкторов и др.

Чтобы получить конкретный конструктор, вызовите метод `getConstructor` с типами аргументов конструктора, как я делал ранее. Аналогично к определенному методу можно получить доступ, вызвав `getMethod` и передав его имя, а также типы параметров.

В вызове `getMethod("toExternalForm")` не указывались никакие типы, поскольку метод не имеет аргументов. Вот `URL.openConnection(Proxy)`, который принимает `Proxy` в качестве параметра:

```
Class<?> urlClass = Class.forName("java.net.URL");
Method openConnectionMethod = urlClass
   .getMethod("openConnection", Proxy.class);
```

Экземпляры, возвращаемые вызовами `getConstructor` и `getMethod`, имеют тип `Constructor` и `Method` соответственно. Для вызова базового члена класса они предлагают такие методы, как `Constructor::newInstance` и `Method::invoke`. Интересной деталью последнего является то, что нужно передать экземпляр, для которого метод должен быть вызван в качестве первого аргумента. Другие аргументы будут переданы вызываемому методу.

Продолжая пример `openConnection`:

```
openConnectionMethod.invoke(url, someProxy);
```

Если вы хотите вызвать статический метод, то можно проигнорировать аргумент экземпляра, передав в него значение `null`.

В дополнение к классу, конструктору и методу, существует также поле, которое обеспечивает доступ для чтения и записи к полям экземпляра. Вызов `get` с экземпляром возвращает значение, которое поле имеет в этом экземпляре, метод `set` устанавливает указанное значение в выбранном экземпляре.

Класс `URL` имеет поле экземпляра `protocol` типа `String`, для URL http://codefx.org он будет содержать `"http"`. Поскольку это приватный код, такой код не будет компилироваться:

```
URL url = new URL("http://codefx.org");
// нет доступа к приватному полю -> ошибка компиляции
url.protocol = "https";
```

Вот как можно сделать то же самое с помощью рефлексии:

```
// `Class<?> urlClass` и `Object url` те же, что и раньше
Field protocolField = urlClass.getDeclaredField("protocol");
Object oldProtocol = protocolField.get(url);
protocolField.set(url, "https");
```

Данный код хоть и компилируется, но все же приводит к `IllegalAccessException` при вызове `get`, поскольку поле `protocol` является приватным. Но это не должно вас останавливать.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Б.2. Взлом API с помощью setAccessible

Одним из важных вариантов использования рефлексии всегда было проникновение в API путем доступа к приватным типам, методам и полям. Это называется *глубокой рефлексией*. Разработчики задействуют ее для доступа к данным, которые API не делает доступными, чтобы обойти ошибки в их зависимостях путем изменения внутреннего состояния и динамически заполнить экземпляры правильными значениями — например, это делает Hibernate.

Для глубокой рефлексии не нужно ничего делать, кроме вызова `setAccessible(true)` для экземпляра `Method`, `Constructor` или `Field` перед его использованием:

```
// `Class<?> urlClass` и `Object url` те же, что и раньше
Field protocolField = urlClass.getDeclaredField("protocol");
protocolField.setAccessible(true);
Object oldProtocol = field.get(url);
protocolField.set(instance, "https");
```

Одна из проблем при переходе на систему модулей состоит в том, что она забирает сверхспособности рефлексии, а это означает гораздо большую вероятность неудачи вызовов `setAccessible`. Подробнее о данной проблеме и о том, как ее исправить, см. в главе 12.

Б.3. Аннотации помечают код для рефлексии

Аннотации — важная часть рефлексии. На самом деле аннотации *ориентированы* на нее. Они предназначены для предоставления метаданных, доступ к которым можно получить во время выполнения, а далее те используются для формирования поведения программы. `@Test` в JUnit и `@Controller` с `@RequestMapping` в Spring — яркие примеры.

Все важные типы, связанные с рефлексией, такие как `Class`, `Field`, `Constructor`, `Method` и `Parameter`, реализуют интерфейс `AnnotatedElement`. Его Javadoc содержит подробное объяснение того, как аннотации могут относиться к этим элементам (непосредственно присутствующим, косвенно представленным или связанным). Однако его простейшая форма заключается в следующем: метод `getAnnotations` возвращает аннотации, которые присутствуют в этом элементе, в виде массива экземпляров `Annotation`, чьи члены могут быть доступны.

Но в контексте системы модулей то, как вы или фреймворки зависите от аннотаций процессов, менее важно, чем факт их работы только с рефлексией. Это означает, что любой класс, который вы видите и на котором есть какие-то аннотации, в какой-то момент будет рефлексирован, — и если он находится в модуле, то не обязательно будет работать «из коробки».

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

B Наблюдение за JVM с унифицированным ведением журнала

Java 9 представил унифицированную архитектуру ведения журнала. Он передает сообщения, которые JVM генерирует благодаря единому механизму, и позволяет выбрать, какие сообщения показывать, с помощью сложного параметра командной строки `-Xlog`.

Его можно задействовать для наблюдения за поведением JVM, отладки приложения, если оно неисправно, или для поиска возможных улучшений производительности. Как вы знаете из собственных проектов, ведение журнала имеет широкую и аморфную область применения, поэтому я не буду объяснять его с помощью одного варианта использования, а вместо этого рассмотрю механизм в целом.

Использование `-Xlog` может поначалу немного пугать, но мы рассмотрим его шаг за шагом, изучив каждый аспект параметра. Здесь мы рассмотрим механизм в целом — в подразделе 5.3.6 показано, как задействовать его для отладки модульного приложения.

ПРИМЕЧАНИЕ

Данный механизм универсален в JVM и имеет гораздо больше способов применения, чем просто мониторинг системы модулей. Загрузка классов, сборка мусора, взаимодействие с операционной системой, многопоточность — можно проанализировать все перечисленное и многое другое, используя правильные флаги. Обратите внимание: это не относится ни к сообщениям JDK, таким как журналы Swing, ни к сообщениям вашего приложения, а касается исключительно самой JVM.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

B.1. Что такое унифицированное ведение журнала

Внутренняя инфраструктура унифицированного ведения журнала JVM аналогична другим фреймворкам ведения журнала, таким как Java Util Logging, Log4j и Logback, которые вы, возможно, использовали для приложений. Она генерирует текстовые сообщения, присоединяет некую метаинформацию, например теги (описывающие исходную подсистему), уровень журнала (описывающий важность сообщения) и отметки времени, и выводит их где-либо. Вывод журнала можно настроить в соответствии с вашими потребностями.

Определение: -Xlog

Параметр `java -Xlog` активирует ведение журнала. Это единственный флаг относительно данного механизма — любая дальнейшая конфигурация немедленно добавляется к самому параметру. Вот что можно настроить для ведения журнала:

- какие сообщения регистрировать (по тегу и/или по уровню журнала);
- какую информацию включать (например, метки времени и идентификаторы процессов);
- какой вывод использовать (к примеру, в файл).

Остальная часть этого приложения рассматривает каждый пункт по очереди.

Прежде чем делать что-либо еще, посмотрим, какие сообщения выдает `-Xlog` (рис. B.1). Выполните `java –Xlog` и посмотрите на выходные данные, которых много. (Вы не предоставили Java достаточно данных для запуска приложения, вследствие чего в нем перечислены все параметры. Чтобы избавиться от этой стены текста, я запускаю ее с `-version`, которая выводит текущую версию Java.)

В одном из первых сообщений говорится, что виртуальная машина HotSpot начинает свою работу:

```
$ java -Xlog -version

# несколько сообщений опущено
> [0.002s][info][os ] HotSpot is running with glibc 2.23, NPTL 2.23
# множество сообщений опущено
```

Оно показывает, насколько долго работает JVM (2 мс), уровень журнала сообщения (информация), его теги (только `os`) и фактическое сообщение. Посмотрим, как повлиять на эти детали.

B.2. Определяем, какие сообщения должны отображаться

Можно применить уровень журнала и теги, чтобы точно установить сферу отображенного журналами, определив пары `<tag-set>=<level>`, которые называются *селекторами*. Все теги могут быть выбраны одновременно друг с другом, и уровень

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

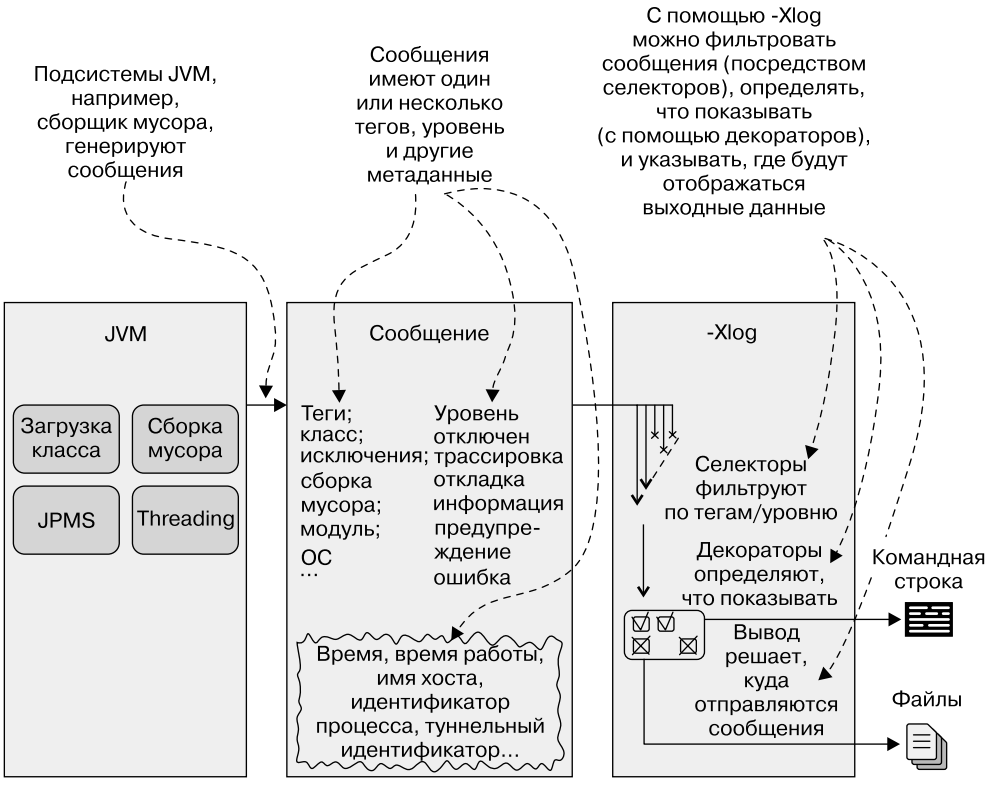


Рис. B.1. Многие подсистемы JVM *(слева)* генерируют сообщения *(в центре)*, а параметр -Xlog можно использовать для настройки того, какие сообщения нужно видеть, какую информацию они включают и где отображаются *(справа)*

является необязательным параметром и по умолчанию установлен в `info`. Использовать это можно следующим образом:

```
$ java -Xlog:all=warning -version

# нет сообщений журнала; отлично, обошлось без предупреждений!
```

Попробуем другой тег и уровень:

```
$ java -Xlog:logging=debug -version

> [0.034s][info][logging] Log configuration fully initialized.
> [0.034s][debug][logging] Available log levels:
    off, trace, debug, info, warning, error
> [0.034s][debug][logging] Available log decorators: [...]
> [0.034s][debug][logging] Available log tags: [...]
> [0.034s][debug][logging] Described tag combinations:
> [0.034s][debug][logging] logging: Logging for the log framework itself
> [0.034s][debug][logging] Log output configuration:
> [0.034s][debug][logging] #0: stdout [...]
> [0.034s][debug][logging] #A: stderr [...]]
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Удачный выстрел! Мне пришлось урезать вывод, поскольку его слишком много, но поверьте, в таких сообщениях куча полезной информации. Однако не обязательно идти по этому пути: `-Xlog:help` показывает ту же информацию, но она более красиво отформатирована (как вы увидите позже).

Удивительная деталь (по крайней мере на первый взгляд) в том, что сообщения соответствуют селектору, только если их теги *точно* соответствуют заданным. Дан *один* тег? Множество тегов? Да, селектор может называть несколько тегов, объединяя их с помощью знака `+`. Тем не менее сообщение должно содержать именно те, которые нужно было выбрать.

Следовательно, использование `gc` (для сборки мусора) в сравнении с `gc+heap`, например, должно выбирать разные сообщения. Это действительно так:

```
java -Xlog:gc -version

[0.009s][info][gc] Using G1

java -Xlog:gc+heap -version

[0.006s][info][gc,heap] Heap region size: 1M
```

Можно определить несколько селекторов одновременно — они просто должны быть разделены запятыми:

```
java -Xlog:gc,gc+heap -version

[0.007s][info][gc,heap] Heap region size: 1M
[0.009s][info][gc     ] Using G1
```

Используя данную стратегию, неудобно получать все сообщения, содержащие конкретный флаг. К счастью, есть более простой способ сделать это: подстановочный знак *, который можно использовать с одним тегом для определения селектора, соответствующего всем сообщениям, содержащим данный тег:

```
java -Xlog:gc*=debug -version
[0.006s][info][gc,heap] Heap region size: 1M
[0.006s][debug][gc,heap] Minimum heap 8388608
 Initial heap 262144000 Maximum heap 4192206848
# опущено около двух десятков сообщений
[0.072s][info ][gc,heap,exit        ] Heap
# опущено несколько сообщений, показывающих окончательную статистику GC
```

Используя ведение журнала и селекторы, вы можете познакомиться с подсистемой JVM, пройдя три простых шага.

1. Найдите интересные теги в выводе `java -Xlog:help`.
2. Используйте их с `-Xlog:${`*`тег_1`*`}*,${`*`тег_2`*`}*,${`*`тег_n`*`}*`, чтобы отобразить все информационные сообщения, которые были помечены любым из них.
3. Выборочно переключайтесь на более низкие уровни журнала с помощью `-Xlog:${`*`тег`*`}*=debug`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Таким образом определяются сообщения, которые будут показаны. Теперь посмотрим, куда их можно вывести.

B.3. Определяем, куда вывести сообщения

По сравнению с нетривиальными селекторами, настройка вывода проста. Вы помещаете ее после селекторов (разделенных двоеточием), и она имеет три возможных значения:

- `stdout` — вывод по умолчанию. В консоли это окно терминала, если нет перенаправления. В IDE результат часто отображается на собственной вкладке или панели;
- `stderr` — вывод ошибок по умолчанию. В консоли это окно терминала, если нет перенаправления. В IDE он обычно отображается на той же вкладке/панели, что и стандартный вывод, но печатается красным цветом;
- `file=<filename>` — определяет файл для передачи всех сообщений. Добавление `file=` не является обязательным.

В отличие от обычных фреймворков ведения журналов, к сожалению, невозможно использовать два варианта вывода одновременно.

Поместить все отладочные сообщения в файл `application.log` можно следующим образом:

```
java -Xlog:all=debug:file=application.log -version
```

Доступны дополнительные параметры вывода, позволяющие развернуть файл журнала в зависимости от размера файла и количества файлов, которые нужно развернуть.

B.4. Определяем, что выводить в сообщениях

Ранее я уже говорил, что каждое сообщение состоит из текста и метаданных. Какую из этих дополнительных частей информации выведет JVM, можно настроить, выбрав *декораторы*, которые перечислены в табл. B.1. Они располагаются после выведенного сообщения и следующего за ним двоеточия.

Допустим, вы хотите вывести в консоль метку времени, время безотказной работы в миллисекундах и идентификатор потока для всех сообщений отладки сборки мусора. Вот как это можно сделать:

```
java -Xlog:gc*=debug:stdout:time,uptimemillis,tid -version

# опущенные сообщения
[2017-02-01T13:10:59.689+0100][7ms][18607] Heap region size: 1M
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Таблица B.1. Декораторы, доступные для опции -Xlog. Информация всегда выводится в данном порядке. Описания основаны на документации

Параметр	Описание
level	Уровень, связанный с сообщением журнала
pid	Идентификатор процесса
tags	Набор тегов, связанный с сообщением журнала
tid	Идентификатор потока
time	Текущее время и дата в формате ISO-8601
timemillis	То же значение, которое генерируется System.currentTimeMillis()
timenanos	То же значение, которое генерируется System.nanoTime()
uptime	Время с момента запуска JVM в секундах (например, 6,567 с)
uptimemillis	Миллисекунды с момента запуска JVM
uptimenanos	Наносекунды с момента запуска JVM

B.5. Настройка всего конвейера регистрации

Формально опция `-Xlog` имеет такой синтаксис:

```
-Xlog:<selectors>:<output>:<decorators>:<output-options>
```

Каждый из параметров, следующих за `-Xlog`, является необязательным, но если вы используете один, то необходимо задействовать все остальные, стоящие перед ним. Селекторы — пары наборов тегов и уровней журнала. Эта часть также называется *what-expression*, термин, с которым вы столкнетесь в случае синтаксически неправильной настройки. Можно определить целевое местоположение для сообщений журнала с `output` (короче говоря, окно терминала или файл журнала) и использовать декораторы, чтобы установить, какую информацию должны содержать сообщения. (И да, досадно, что механизм вывода и дальнейшие параметры вывода разделены, между ними есть декораторы.)

Для получения дополнительной информации см. онлайн-документацию по адресу http://mng.bz/K1Gj или вывод `java -Xlog:help`:

```
java -Xlog:help

-Xlog Usage: -Xlog[:[what][:[output][:[decorators][:output-options]]]]
        where 'what' is a combination of tags and levels on the form
            tag1[+tag2...][*][=level][,...]
        Unless wildcard (*) is specified, only log messages tagged with
            exactly the tags specified will be matched.

Available log levels:
    off, trace, debug, info, warning, error

Available log decorators:
    time (t), utctime (utc), uptime (u), timemillis (tm), uptimemillis (um),
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

```
    timenanos (tn), uptimenanos (un), hostname (hn), pid (p), tid (ti),
    level (l), tags (tg)
    Decorators can also be specified as 'none' for no decoration.

Described tag combinations:
    logging: Logging for the log framework itself

Available log tags:
    [... много, много тегов ... ]
    Specifying 'all' instead of a tag combination matches all tag
     combinations.

Available log outputs:
    stdout, stderr, file=<filename>
    Specifying %p and/or %t in the filename will expand to the JVM's PID and
    startup timestamp, respectively.

Some examples:
    [... несколько примеров, которые могут пригодиться ... ]
```

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Г

Анализ зависимостей проекта с помощью JDeps

JDeps — это *инструмент анализа зависимостей Java*, инструмент командной строки, который обрабатывает байт-код Java — файлы `.class` или JAR, которые их содержат, — и анализирует статически объявленные зависимости между классами. Результаты можно отфильтровать различными способами и агрегировать на уровне пакета или JAR. JDeps также полностью осведомлен о системе модулей.

В целом, это полезный инструмент для изучения различных, иногда туманных графов, о которых я так много говорю в книге. Более того, у него есть конкретные приложения для миграции и модуляризации проекта, такие как анализ его статических зависимостей от внутренних API JDK (см. подраздел 7.1.2), перечисление разделенных пакетов (см. подраздел 7.2.5) и составление дескрипторов модулей (см. подраздел 9.3.2).

Чтобы его изучить, я призываю вас проследовать дальше, предпочтительно задействуя один из *ваших* проектов. Будет проще всего, если у вас есть JAR-файл проекта и папка со всеми его переходными зависимостями. В Maven можно достичь последнего с помощью `copy-dependencies` плагина *maven-dependency-plugin*. В Gradle можно использовать задачу `Copy`, настроив `from` как файл `configurations.compile` или `configurations.runtime`. Для деталей пригодится быстрый поиск.

В качестве примера проекта я выбрал Scaffold Hunter:

> Scaffold Hunter — это инструмент с открытым исходным кодом на основе Java для визуального анализа наборов данных с акцентом на данных из естественных наук, нацеленный на интуитивный доступ к большим и сложным наборам данных. Инструмент предлагает различные виды, например график, дендрограмму и диаграмму, а также методы анализа, например, для кластеризации и классификации.
>
> *scaffoldhunter.sourceforge.net*

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Я скачал Zip-файл версии 2.6.3 и скопировал все зависимости в `libs`. При отображении вывода я сокращаю `scaffoldhunter` (в именах пакетов) и `scaffoldhunter` (в именах файлов) до `sh`, чтобы сделать его короче.

Г.1. Знакомство с JDeps

Начнем с знакомства с JDeps: где его найти, как получить первые результаты и куда обратиться за помощью. Исполняемый файл JDeps `jdeps` можно найти в папке `bin` вашего JDK, начиная с Java 8. Работать с ним проще всего, если он доступен в командной строке, для чего вам может потребоваться выполнить некоторые шаги настройки, специфичные для выбранной ОС. Убедитесь, что `jdeps --version` работает и показывает, что используется самая последняя версия.

Следующий шаг — захват JAR и высвобождение его с помощью JDeps. При использовании без дополнительных параметров командной строки он сначала перечислит модули JDK, от которых зависит код, включая упоминание о том, что они не найдены для всего кода, который не является ни частью JAR, ни частью JDK. Далее следует список зависимостей уровня пакета, организованный как ${*пакет*} -> ${*пакет*} ${*модуль/JAR*}.

Вызов `jdeps scaffold-hunter-2.6.3.jar` приводит к следующему ошеломляющему результату. Вы можете видеть, что Scaffold Hunter зависит от модулей *java.base* (естественно), *java.desktop* (это приложение Swing), *java.sql* (наборы данных хранятся в базах данных SQL) и некоторых других. За этим списком зависимостей следует длинный список зависимостей пакетов, слишком сложный для восприятия:

```
$ jdeps scaffold-hunter-2.6.3.jar

# помните, что "sh" — сокращение от "scaffold-hunter"
# (в именах файлов) и "scaffoldhunter" (в именах пакетов)
> sh-2.6.3.jar -> java.base
> sh-2.6.3.jar -> java.datatransfer
> sh-2.6.3.jar -> java.desktop
> sh-2.6.3.jar -> java.logging
> sh-2.6.3.jar -> java.prefs
> sh-2.6.3.jar -> java.sql
> sh-2.6.3.jar -> java.xml
> sh-2.6.3.jar -> not found
>    edu.udo.sh -> com.beust.jcommander  not found
>    edu.udo.sh -> edu.udo.sh.data       sh-2.6.3.jar
>    edu.udo.sh -> edu.udo.sh.gui        sh-2.6.3.jar
>    edu.udo.sh -> edu.udo.sh.gui.util   sh-2.6.3.jar
>    edu.udo.sh -> edu.udo.sh.util       sh-2.6.3.jar
>    edu.udo.sh -> java.io               java.base
>    edu.udo.sh -> java.lang             java.base
>    edu.udo.sh -> javax.swing           java.desktop
>    edu.udo.sh -> org.slf4j             not found
# опущены многие другие зависимости пакета
```

Модули JDK, от которых зависит проект

Зависимости пакетов внутри JAR-файлов и между ними

«Не найдено» означает, что зависимости не были найдены, и это неудивительно, поскольку я не указал JDeps, где их искать

Теперь пришло время настроить вывод с помощью различных параметров. Можно перечислить их с помощью `jdeps -h`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Г.2. Добавление зависимостей в анализ

Важным аспектом JDeps является то, что он позволяет анализировать зависимости, как если бы они были частью кода. Первый шаг к этой цели — размещение их в пути к классам с помощью `--class-path`, но это только позволяет JDeps следовать по путям в JAR-файлы зависимостей и избавляет вас от индикаторов `not found`. Чтобы проанализировать зависимости, нужно заставить JDeps рекурсировать в них с помощью `-recursive` или `-r`.

В целях добавления зависимостей Scaffold Hunter я выполнил JDeps с параметрами `--class-path 'libs/*'` и `-recursive`, результат чего можно увидеть далее. В этом конкретном случае вывод начинается с нескольких предупреждений о разделенных пакетах, которые я сейчас пока проигнорирую. Следующие зависимости модуля/JAR и пакета такие же, как и раньше, но теперь все они найдены, поэтому их гораздо больше:

```
$ jdeps -recursive
    --class-path 'libs/*'
    scaffold-hunter-2.6.3.jar

# опущены предупреждения о разделенном пакете
# опущены некоторые зависимости модуля/JAR
> sh-2.6.3.jar -> libs/commons-codec-1.6.jar
> sh-2.6.3.jar -> libs/commons-io-2.4.jar
> sh-2.6.3.jar -> libs/dom4j-1.6.1.jar
> sh-2.6.3.jar -> libs/exp4j-0.1.38.jar
> sh-2.6.3.jar -> libs/guava-18.0.jar
> sh-2.6.3.jar -> libs/heaps-2.0.jar
> sh-2.6.3.jar -> libs/hibernate-core-4.3.6.Final.jar
> sh-2.6.3.jar -> java.base
> sh-2.6.3.jar -> java.datatransfer
> sh-2.6.3.jar -> java.desktop
> sh-2.6.3.jar -> java.logging
> sh-2.6.3.jar -> java.prefs
> sh-2.6.3.jar -> java.sql
> sh-2.6.3.jar -> java.xml
> sh-2.6.3.jar -> libs/javassist-3.18.1-GA.jar
> sh-2.6.3.jar -> libs/jcommander-1.35.jar
# опущено больше зависимостей модуля / JAR
>    edu.udo.sh -> com.beust.jcommander  jcommander-1.35.jar
>    edu.udo.sh -> edu.udo.sh.data       sh-2.6.3.jar
>    edu.udo.sh -> edu.udo.sh.gui        sh-2.6.3.jar
>    edu.udo.sh -> edu.udo.sh.gui.util   sh-2.6.3.jar
>    edu.udo.sh -> edu.udo.sh.util       sh-2.6.3.jar
>    edu.udo.sh -> java.io               java.base
>    edu.udo.sh -> java.lang             java.base
>    edu.udo.sh -> javax.swing           java.desktop
>    edu.udo.sh -> org.slf4j             slf4j-api-1.7.5.jar
# опущено много, много других зависимостей пакетов
```

Больше нет «ненайденных» зависимостей JAR

Больше нет «ненайденных» источников для зависимостей пакетов

Это делает вывод еще более ошеломляющим, и, следовательно, самое время взглянуть на то, что вообще можно понять из такого большого количества данных.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Г.3. Настройка вывода JDeps

Существуют различные способы настройки вывода JDeps. Возможно, лучший вариант для использования в первом анализе любого проекта — `-summary` или `-s`, которые показывают только зависимости между JAR-файлами, как показано здесь:

```
 $ jdeps -summary -recursive
--class-path 'libs/*'
scaffold-hunter-2.6.3.jar

# опущены предупреждения о разделенном пакете
# опущены некоторые зависимости модуля/JAR
> sh-2.6.3.jar -> libs/javassist-3.18.1-GA.jar
> sh-2.6.3.jar -> libs/jcommander-1.35.jar
> sh-2.6.3.jar -> libs/jgoodies-forms-1.4.1.jar
> sh-2.6.3.jar -> libs/jspf.core-1.0.2.jar
> sh-2.6.3.jar -> libs/l2fprod-common-sheet.jar
> sh-2.6.3.jar -> libs/l2fprod-common-tasks.jar
> sh-2.6.3.jar -> libs/opencsv-2.3.jar
> sh-2.6.3.jar -> libs/piccolo2d-core-1.3.2.jar
> sh-2.6.3.jar -> libs/piccolo2d-extras-1.3.2.jar
> sh-2.6.3.jar -> libs/slf4j-api-1.7.5.jar
> sh-2.6.3.jar -> libs/xml-apis-ext.jar
> sh-2.6.3.jar -> libs/xstream-1.4.1.jar
> slf4j-api-1.7.5.jar -> java.base
> slf4j-api-1.7.5.jar -> libs/slf4j-jdk14-1.7.5.jar
> slf4j-jdk14-1.7.5.jar -> java.base
> slf4j-jdk14-1.7.5.jar -> java.logging
> slf4j-jdk14-1.7.5.jar -> libs/slf4j-api-1.7.5.jar
# опущено больше зависимостей модуля/JAR
```

В табл. Г.1 перечислены фильтры, которые позволяют взглянуть на зависимости под разными углами.

Таблица Г.1. Краткое описание некоторых параметров, которые фильтруют вывод JDeps

Параметр	Описание
--api-only или -apionly	Иногда, особенно при анализе библиотеки, вы заботитесь только об API JAR-файла. С помощью этого параметра рассматриваются лишь типы, упомянутые в подписях публичных и защищенных членов публичных классов
-filter или -f	Затем следует регулярное выражение, которое исключает зависимости от классов, соответствующих регулярному выражению. (Обратите внимание: если не используется -verbose:class, то вывод по-прежнему показывает пакеты)
-filter:archive	Во многих случаях зависимости внутри артефакта не столь интересны. Этот параметр игнорирует их и показывает только зависимости между артефактами
--package или -p	После имени пакета учитываются лишь зависимости от данного пакета, что является отличным способом увидеть все места, где используются эти utils
--regex или -e	За ним следует регулярное выражение, учитываются только зависимости от классов, соответствующих регулярному выражению. (Обратите внимание: если не используется -verbose:class, то вывод по-прежнему показывает пакеты)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Вывод в командной строке — хороший способ изучить детали и углубиться в интересные фрагменты. Хотя он и не предназначен для наиболее интуитивного обзора — диаграммы в этом смысле гораздо более полезны. К счастью, JDeps имеет параметр `--dot-output`, который создает файлы `.dot` для каждого отдельного анализа. Эти файлы представляют собой чистый текст, но создать их можно с помощью других инструментов, таких как Graphviz. В листинге Г.1 и на рис. Г.1 показан пример.

Листинг Г.1. Визуализация зависимостей артефакта

```
$ jdeps -recursive
    --class-path 'libs/*'
    --dot-output dots
    scaffold-hunter-2.6.3.jar
$ dot -Tpng -O dots/summary.dot
```

--dot-output dots указывает Jdeps создавать файлы .dot в папке dots

Graphviz предоставляет команду dot, которая используется здесь для создания summary.dot.png в dots

Файлы .dot и Graphviz

Файл `.dot` представляет собой простой текст и является отличным промежуточным представлением для редактирования. С помощью некоторых регулярных выражений можно, например, удалить модуль `java.base` снизу (делает граф проще) или версии из имен JAR (делает граф стройнее). Чтобы получить дополнительную информацию о Graphviz, посетите сайт graphviz.gitlab.io.

Г.4. Детализация в зависимости от вашего проекта

Если вы хотите увидеть больше деталей, то `-verbose:class` перечисляет зависимости между классами, а не агрегирует их на уровне пакета. Иногда перечисления только прямых зависимостей от пакета или класса недостаточно, поскольку они могут быть не в коде, а скорее в зависимостях. В этом случае может помочь `--inverse` или `-I`. Учитывая определенный пакет или регулярное выражение для поиска, он отслеживает зависимости в обратном порядке, так далеко, насколько далеко они распространяются, при этом перечисляя артефакты по пути. К сожалению, кажется, нет простого способа увидеть результат на уровне классов, а не артефактов.

Если вас интересуют лишь зависимости, представленные в публичном API библиотеки, то здесь можно использовать `--api-only`. При этом рассматриваются только типы, упомянутые в подписях публичных и защищенных членов публичных классов. Есть еще несколько параметров, которые могут помочь в вашем конкретном случае, — как уже упоминалось, их можно перечислить с помощью `jdeps -h`.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

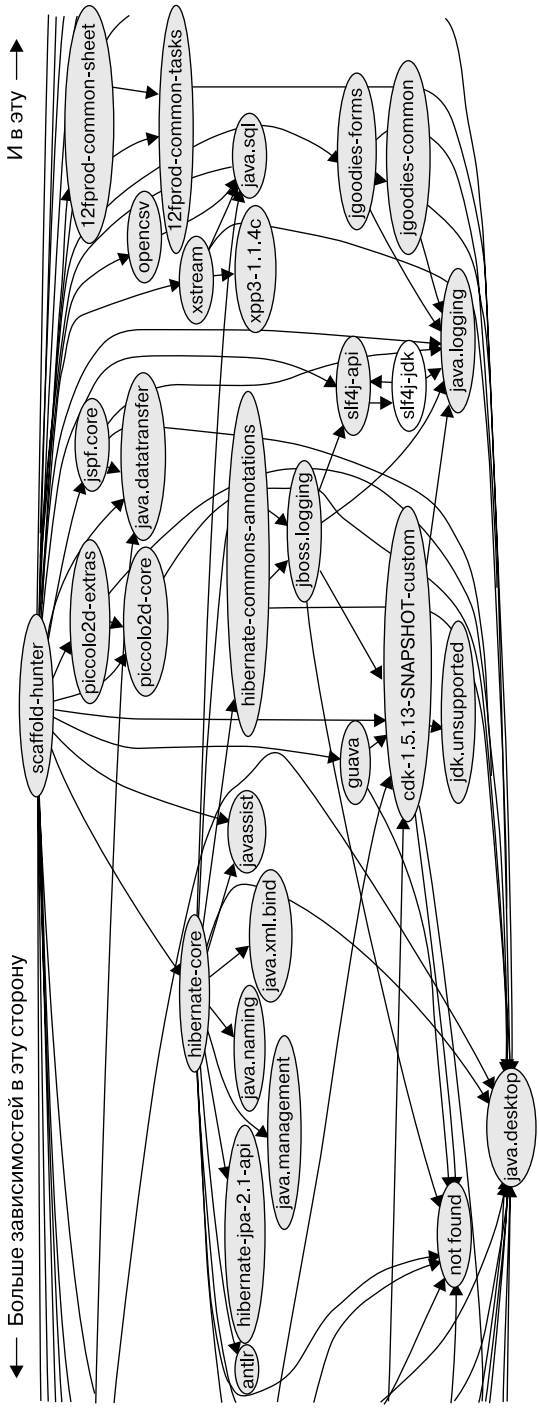


Рис. Г.1. Результат листинга Г.1 — большой, сложный, но все еще доступный граф зависимостей. Здесь представлена всего лишь очищенная его часть. Не беспокойтесь о деталях; вместо этого создайте подобный граф для своего проекта

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Г.5. JDeps понимает модули

Как компилятор и JVM могут работать на более высоком уровне абстракции благодаря системе модулей, так и JDeps может делать то же самое. Путь модуля можно указать с помощью `--module-path` (обратите внимание, что `-p` зарезервирован: это не сокращение для данного параметра), а начальный модуль — с `--module` или `-m`. Оттуда можно проводить те же виды анализов, что и раньше:

```
$ jdeps -summary -recursive
    --module-path mods:libs
    -m monitor

# опущены некоторые зависимости модуля
> monitor -> java.base
> monitor -> monitor.observer
> monitor -> monitor.observer.alpha
> monitor -> monitor.observer.beta
> monitor -> monitor.persistence
> monitor -> monitor.rest
> monitor -> monitor.statistics
> monitor.observer -> java.base
> monitor.observer.alpha -> java.base
> monitor.observer.alpha -> monitor.observer
> monitor.observer.beta -> java.base
> monitor.observer.beta -> monitor.observer
> monitor.persistence -> java.base
> monitor.persistence -> monitor.statistics
> monitor.persistence -> hibernate.jpa
> monitor.rest -> java.base
> monitor.rest -> monitor.statistics
> monitor.rest -> spark.core
> monitor.statistics -> java.base
> monitor.statistics -> monitor.observer
> slf4j.api -> java.base
> slf4j.api -> not found
> spark.core -> JDK removed internal API
> spark.core -> java.base
> spark.core -> javax.servlet.api
> spark.core -> jetty.server
> spark.core -> jetty.servlet
> spark.core -> jetty.util
> spark.core -> slf4j.api
> spark.core -> websocket.api
> spark.core -> websocket.server
> spark.core -> websocket.servlet
# опущено больше зависимостей модуля
```

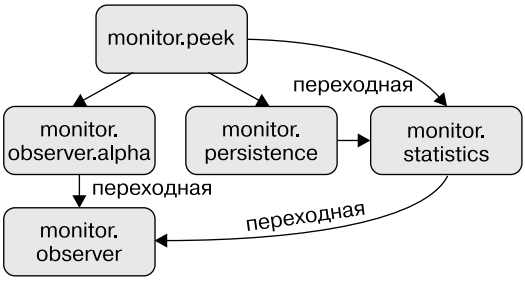
Помимо этого, есть некоторые специфичные для Java 9 и модуля опции. С помощью `--require ${`*модули*`}` можно перечислить все модули, запрашивающие другие именованные модули. Как использовать `--jdk-internals` для анализа проблемных зависимостей проекта, описано в подразделе 7.1.2. В подразделе 9.3.2 объясняется, как `--generate-module-info` и `--generateopen-module` можно использовать для создания первых проектов дескрипторов модулей. Как уже упоминалось, JDeps также

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

всегда будет сообщать обо всех найденных разделенных пакетах — проблема, подробно обсуждаемая в разделе 7.2.

Интересен параметр `--check`, который предоставляет различные перспективы для дескриптора модуля (рис. Г.2).

- Он начинается с вывода фактического дескриптора, за которым следуют два гипотетических.
- Первый из них, описываемый как *предлагаемый дескриптор*, объявляет зависимости от всех модулей, типы которых используются в проверяемом модуле.
- Второй, описанный как *переходный сокращенный граф*, аналогичен первому, но удаляет те зависимости, которые могут быть прочитаны из-за подразумеваемой читабельности (см. раздел 9.1). Это самый маленький набор зависимостей, который обеспечивает надежную конфигурацию.
- Наконец, если модуль объявляет любые квалифицированные экспорты (см. раздел 9.3), то `--check` выведет те, которые не используются в совокупности обозреваемых модулей.




```
jdeps --module-path mods:libs --check monitor.peek

monitor.peek (file:///.../mods/monitor.peek.jar)
  [Module descriptor]
  requires mandated java.base (@9.0.4);
  requires monitor.observer.alpha;
  requires monitor.persistence;
  requires monitor.statistics;
  [Suggested module descriptor for monitor.peek ]
  requires mandated java.base;
  requires monitor.observer;
  requires monitor.observer.alpha;
  requires monitor.persistence;
  [Transitive rediced graph for monitor.peek]
  requires mandated java.base;
  requires monitor.observer.alpha;
  requires monitor.persistence;
```

Рис. Г.2. *Слева* расположен monitor.peek (представлен в подразделе 11.1.1) и его переходные зависимости, часть которых подразумевает читабельность других модулей. *Справа* JDeps предлагает включить monitor.observer в список зависимостей (поскольку на его типы ссылаются напрямую). Кроме того, он перечисляет абсолютный минимальный набор модулей, на которые monitor.peek должен ссылаться для надежной конфигурации, в полной мере используя подразумеваемую читабельность

Гипотетические дескрипторы, которые создает `--check`, также можно просматривать отдельно с помощью двух параметров `--list-deps` и `--list- reduced-deps` соответственно. Они также работают с путем класса и в этом случае ссылаются на безымянный модуль (см. раздел 8.2).

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Приложение А
Ориентация на несколько версий Java с мультиверсионными JAR-файлами

Решение о том, какая версия Java требуется для проекта, никогда не было простым. С одной стороны, вы хотите предоставить пользователям свободу выбора, поэтому было бы неплохо поддерживать несколько основных версий, а не только новейшую. С другой стороны, вам не терпится применить новейшие языковые функции и API. Начиная с Java 9, появилась новая функция JVM — *мультиверсионные* JAR (MB-JAR), помогающие вам примирить эти противоборствующие силы, по крайней мере при некоторых обстоятельствах.

MB-JAR позволяют вам отправлять байт-код для разных версий Java в одном и том же артефакте. Затем можно положиться на JVM для загрузки скомпилированных классов для самой последней поддерживаемой версии. Можно начать с проекта, который успешно выполняется на минимально требуемой версии, и выборочно улучшить его на более новых JVM, используя более гибкие и эффективные API — не прибегая к необходимости повышать базовый уровень проекта.

ВАЖНАЯ ИНФОРМАЦИЯ

Конечно, нужно будет рассмотреть MB-JAR только в том случае, если у вас нет полного контроля над версией JVM, в которой выполняется проект. Это всегда относится к библиотекам и фреймворкам и часто к приложениям для ПК или веб-серверам, которые ваши пользователи размещают самостоятельно. С другой стороны, если вы управляете машинами, на которых работает ваше приложение, то можно использовать более новую JVM и отказаться от сложностей MB-JAR.

Разобравшись со всем вышесказанным, рассмотрим эту удобную новую функцию. Начнем с создания простого MR-JAR, а затем изучим его внутреннюю структуру. Закончим некоторыми рекомендациями о том, когда и как использовать MB-JAR.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Д.1. Создание мультиверсионного JAR

Определение

Мультиверсионные JAR-файлы (MB-JAR) представляют собой специально подготовленные JAR-файлы, которые содержат байт-код для нескольких основных версий Java. Способ загрузки этого байт-кода зависит от версии JVM:

- Java 8 и более ранние версии загружают файлы классов, не зависящие от версии;
- Java 9 и более поздние версии загружают файлы классов, зависящие от версии, если те существуют, или иным образом возвращаются к файлам, не зависящим от версии.

Чтобы подготовиться к MB-JAR, необходимо разделить исходные файлы по целевой версии Java, скомпилировать каждый набор источников для соответствующей версии и поместить полученные файлы `.class` в отдельные папки. При упаковке их с помощью `jar` обычно добавляют файлы базовых классов (напрямую или с `-C`; см. подраздел 4.5.1) и используют новый параметр `--release ${`*`релиз`*`}` для каждого другого набора байт-кодов.

Рассмотрим пример. Допустим, необходимо определить основную версию JVM, которая работает в данный момент. Java 9 предлагает хороший API для таких случаев, поэтому больше не нужно анализировать системное свойство. (Подраздел 6.5.1 дает представление об этом, но подробности здесь неважны.) Развернув MR-JAR, можно использовать данный API при работе на Java 9 или новее.

Гипотетическое приложение имеет два класса, `Main` и `DetectVersion`; и цель состоит в том, чтобы иметь два варианта `DetectVersion`, один для Java 8 и более ранних версий, а другой для Java 9 и более поздних. Эти два варианта должны иметь одно и то же полностью определенное имя (что может затруднить работу с ними в IDE) — предположим, вы помещаете их в две параллельные исходные папки, `src/main/java` и `src/main/java-9`.

На рис. Д.1 показано, как организовать источники, а в листинге Д.1 объясняется, как скомпилировать и упаковать их в MB-JAR. Обратите внимание на два этапа компиляции и отдельные папки вывода. Конечный результат показан на рис. Д.2.

Листинг Д.1. Компиляция и упаковка исходников для разных версий Java в JAR

```
Компилирует код в src/main/java для Java 8
(или более ранних версий) в classes
javac --release 8
    -d classes
    src/main/java/org/codefx/detect/*.java
javac --release 9
    -d classes-9
    src/main/java-9/module-info.java
    src/main/java-9/org/codefx/detect/DetectVersion.java
jar --create
    --file target/detect.jar
    -C classes .
    --release 9
    -C classes-9 .
```

Компилирует код в src/main/java-9 для Java 9 в classes-9

При упаковке байт-кода в JAR-файл, как обычно, упаковывает байт-код по умолчанию из классов

Включает классы специально для Java 9

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

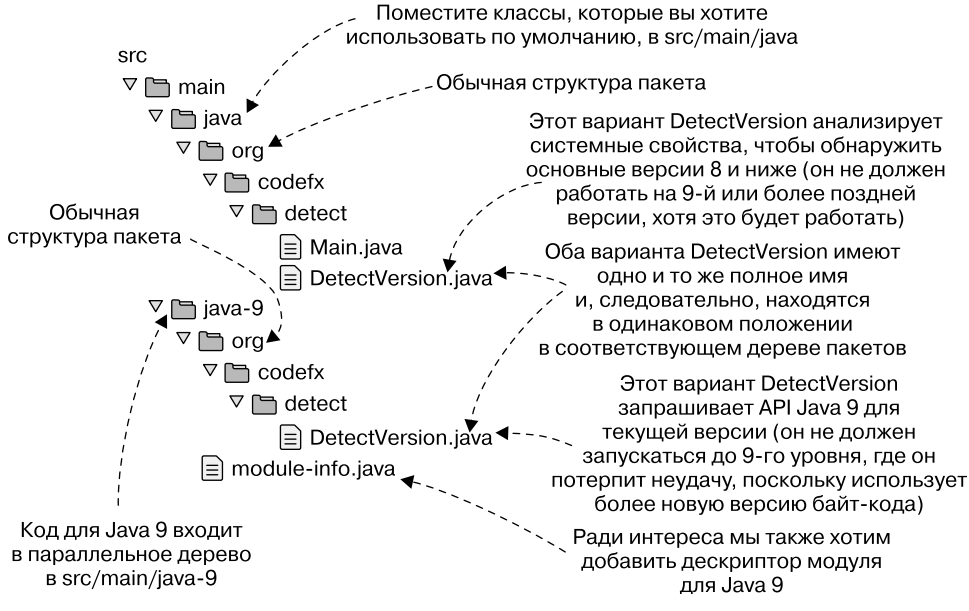


Рис. Д.1. Один из возможных способов выложить исходный код MB-JAR. Наиболее важная деталь — то, что зависимый от версии код, в данном случае DetectVersion, имеет одно и то же полное имя во всех вариантах

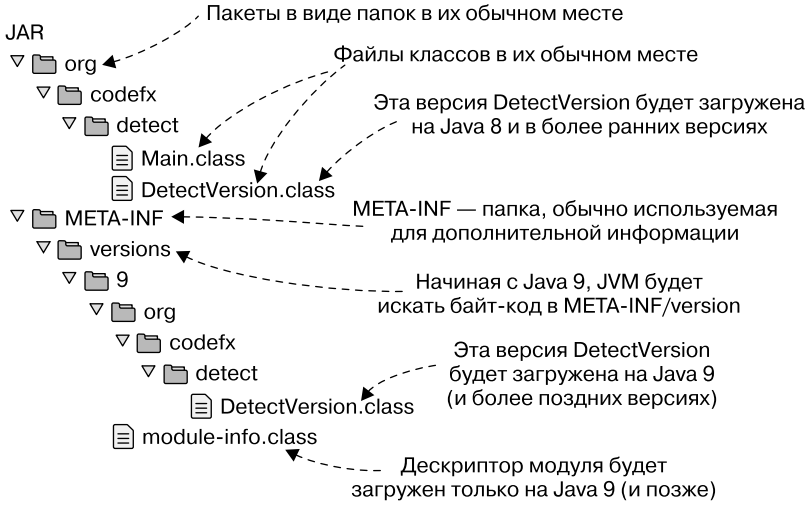


Рис. Д.2. JAR, полученный в результате листинга Д.1

В этом простом примере создаются два варианта `DetectVersion`, один для минимально необходимой Java 8, а другой — для Java 9. Формализация этого в общем случае создания функции с несколькими классами для нескольких версий на удивление сложна и утомительна, так что я избавлю вас от официальной версии. Вместо этого раздел Д.3 демонстрирует практическое правило.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Д.2. Внутренняя работа MB-JAR

Как работает MR-JAR? Это довольно просто: он хранит файлы классов, не зависящих от версии, в своем корневом каталоге (как обычно) и файлах, зависящих от версии, в `META-INF/version/${`*версия*`}`.

ВАЖНАЯ ИНФОРМАЦИЯ

JVM версии 8 и более ранних версий ничего не знают о META-INF/version и загружают классы из структуры пакета в корень JAR. Следовательно, невозможно различить версии до 9.

Однако более новые JVM сначала изучают версии META-INF/ и только в том случае, если они не находят там класс, — в корне JAR. Они выполняют поиск в обратном направлении по своей собственной версии, то есть Java 10 JVM ищет код в META-INF/version/10, затем в META-INF/version/9 и в корневом каталоге. Таким образом, эти JVM прикрывают файлы классов, не зависящие от версии, новейшими версиями, которые они поддерживают.

В дополнение к папкам в `META-INF/version`, MB-JAR также можно распознать, просмотрев открытый текстовый файл `META-INF/MANIFEST.MF`: в MB-JAR манифест содержит запись `Multi-Release: true`.

Д.3. Рекомендации по использованию

Теперь, когда вы знаете, как создавать MR-JAR и принципы их работы, я хочу дать вам несколько рекомендаций о способах извлечения из них максимальной пользы. Точнее, я дам вам советы по следующим темам:

- как организовать исходный код;
- как организовать байт-код;
- когда использовать MR-JAR.

Д.3.1. Организация исходного кода

ВАЖНАЯ ИНФОРМАЦИЯ

Я предлагаю два руководства при организации исходного кода для MB-JAR.

- Код для самой старой поддерживаемой версии Java помещается в корневой каталог проекта по умолчанию: например, src/main/java, а не src/main/java-X.
- Код в этой исходной папке завершен, то есть его можно скомпилировать, протестировать и развернуть как есть, без дополнительных файлов из исходных деревьев, зависящих от версии, таких как src/main/java-X. (Обратите внимание: если вы предлагаете функцию, работающую только на более новой версии Java, то наличие класса, который лишь выдает ошибки, указывающие, что Operation not supported before Java X считается завершенным. Я рекомендую не пропускать такие моменты, иначе они приводят к неинформативной NoClassDefFoundError.)

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Это не технические требования; ничто не мешает вам ориентироваться на Java 11 и помещать половину кода в `src/main/java`, а другую половину или даже всю часть — в `src/main/java-11`. Но это только вызовет путаницу.

Придерживаясь рекомендаций, вы сохраняете макет исходного дерева максимально простым. Любой человек или инструмент, который изучает его, видит полностью функционирующий проект, нацеленный на требуемую версию JVM. Затем зависящие от версии исходные деревья выборочно улучшают этот код для более новых версий.

Как проверить, правильно ли вы поняли? Я уже говорил ранее, что формальное описание сложное, поэтому вот оно — обещанное правило. Для определения того, работает ли конкретный макет, мысленно (или фактически) выполните следующие шаги.

1. Скомпилируйте и протестируйте независимое от версии дерево исходных текстов на самой старой поддерживаемой версии Java.
2. Для каждого дополнительного исходного дерева:
   - переместите зависимый от версии код в независимое от версии дерево, заменив файлы, в которых они имеют одно и то же полное имя;
   - скомпилируйте и протестируйте дерево на более новой версии.

Если это работает, то вы все поняли правильно.

Конечно, ваши инструменты также должны работать с выбранным макетом источника. К сожалению, на момент написания этой книги IDE и большинство инструментов сборки не имели хорошей поддержки для данного макета и есть вероятность того, что вам придется пойти на компромисс. В качестве альтернативного решения рассмотрите возможность создания отдельных проектов для каждой версии Java.

Д.3.2. Организация байт-кода

ВАЖНАЯ ИНФОРМАЦИЯ

Прямой путь от этой исходной древовидной структуры ведет к моему предложению об организации байт-кода в JAR:

- байт-код для самой старой поддерживаемой версии Java входит в корень JAR, то есть он не добавляется после --release;
- байт-код в корне JAR завершен; это значит, что он может быть выполнен без дополнительных файлов из META-INF/versions.

Еще раз: это не технические требования, но они гарантируют, что каждый, кто изучает корень JAR, видит полностью функционирующий проект, скомпилированный для требуемой версии JVM, с выборочными улучшениями для более новых JVM в версиях `META-INF/versions`.

Д.3.3. Когда использовать MB-JAR

Как MB-JAR помогают решить дилемму выбора необходимой версии Java? Для начала, и это очевидно, подготовка MB-JAR добавляет немало сложностей.

- Среда IDE и инструмент сборки должны быть настроены соответствующим образом, чтобы обеспечить простую работу с исходными файлами с одинако-

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

вым полностью определенным именем, которые скомпилированы для разных версий Java.

- Необходимо синхронизировать несколько вариантов одного и того же исходного файла, чтобы они содержали один и тот же общедоступный API.
- Модульное тестирование становится более сложным, поскольку в конечном итоге можно написать тесты, которые запускаются или проходят только на определенных версиях JVM.
- Интеграционное тестирование становится более громоздким, так как приходится рассматривать возможность проверки полученного артефакта на каждой версии Java, для которой MB-JAR содержит байт-код.

ВАЖНАЯ ИНФОРМАЦИЯ

Это означает необходимость тщательно продумать, хотите ли вы создать MB-JAR. Чтобы идти по этому пути, должна наличествовать значительная мотивация. (Может быть, в конце концов вы сможете поднять необходимую версию Java.)

Кроме того, MR-JAR не подходят для использования удобных новых языковых функций. Как вы уже видели, нужны два варианта задействованных исходных файлов, и нет никакого веского аргумента в пользу удобства, если требуется сохранить исходный файл с неудобным вариантом. Языковые функции также быстро проникают в кодовую базу, что приводит к появлению множества дублирующихся классов. Это не очень хорошая идея.

С другой стороны, API — это лучшее место для MR-JAR. Java 9 представила ряд новых API, которые вносят большую гибкость и/или лучшую производительность в существующие варианты использования:

- обнаружение версии JVM с помощью `Runtime.Version` вместо анализа системных свойств (см. подраздел 6.5.1);
- анализ стека вызовов с помощью API обхода стека вместо создания `Throwable` (эта книга не охватывает данный API, но разработчики вашей среды ведения журналов уже используют его);
- замена рефлексии обработчиками переменных (см. подраздел 12.3.2).

Если вы хотите задействовать более новый API в более новой версии Java, то все, что нужно сделать, — это инкапсулировать прямые обращения к нему в выделенном классе-обертке, а затем реализовать два его варианта: один с помощью старого API, другой — с помощью нового. Если вы смирились со сложностями, изложенными ранее, то это будет совсем не трудно.

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/

Николай Парлог

Система модулей Java

Перевел с английского *А. Павлов*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 08.10.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 37,410. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Больше книг по языку Java по ссылке https://coderbooks.ru/books/java/